



# Introducción a la Programación

Clases teóricas

por Pablo E. “Fidel” Martínez López

## 8. Registros y variantes





# Repaso



- **Programar es comunicar** (con máquinas y personas)
  - Estrategia de solución (división en subtarefas)
  - Legibilidad (elección de nombres, indentación)
    - **CONTRATOS:** Propósito, parámetros y precondiciones
- **Programas** (texto con diversos elementos)
  - **Comandos:** describen acciones
  - **Expresiones:** describen información
    - **Tipos:** clasifican expresiones



- **Comandos**

- Primitivos y secuencia
- PROCEDIMIENTOS (con y sin parámetros)
- Repetición simple
- Alternativa condicional
- Repetición condicional
- Asignación de variables



- **Expresiones**

- Valores literales y expresiones primitivas
- Operadores
  - numéricos, de enumeración, de comparación, lógicos
- Alternativa condicional en expresiones
- **FUNCIONES**
  - (con y sin parámetros, con y sin procesamiento)
- Parámetros (como datos)
- Variables (como datos)



- **Tipos de datos**

- permiten clasificar expresiones
- en Gobstones, por ahora, son cuatro
  - colores, direcciones, números y valores de verdad
- toda expresión tiene un tipo
- los parámetros deben especificar qué tipo de expresiones aceptan



# Registros

- Gobstones solo tiene como primitivos 4 tipos
  - Colores, Direcciones, Números y Booleanos
- ¿Cómo definimos entonces una carta?
  - Una carta está compuesta por dos partes
    - Palo y número



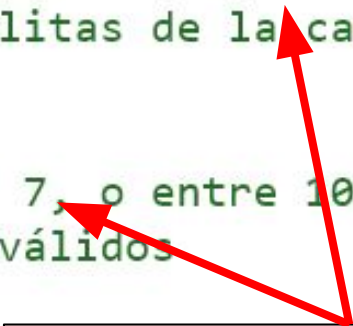
Este bloque describe la carta





- ¿Cómo definimos entonces una carta?
  - Una carta está compuesta por dos partes
    - Palo y Número
  - Si usamos variables, es complicado...

```
procedure PonerCarta__FEO(númeroDeLaCarta, paloDeLaCarta) {  
  /* PROPÓSITO: pone la codificación de bolitas de la carta  
    dada por los parámetros  
  PRECONDICIÓN:  
    * el númeroDeLaCarta está entre 1 y 7, o entre 10 y 12  
    * el paloDeLaCarta es uno de los 4 válidos  
  PARÁMETROS:  
    * númeroDeLaCarta es un Número  
    * paloDeLaCarta es un Palo  
      (como sea que se represente)  
  */  
  Poner__Veces(Azul, 3)  
  Poner__Veces(Negro, códigoDeCarta(númeroDeLaCarta, paloDeLaCarta))  
}
```



Dos parámetros, muchas precondiciones

- ¿Cómo definimos entonces una carta?
  - Una carta está compuesta por dos partes
    - Palo y Número
  - ...¿sería mejor tener un tipo Carta!

```
procedure PonerCarta_(carta) {  
  /* PROPÓSITO: pone la codificación de la carta dada usando bolitas  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: carta es de tipo Carta  
  */  
  Poner__Veces(Azul, 3)  
  Poner__Veces(Negro, códigoDeCarta(númeroDe(carta), paloDe(carta)))  
}
```

¿Pero cómo se define el  
tipo Carta?



- Las cartas son un ejemplo de ***dato con estructura***
  - Son datos que tienen más de una parte
  - Podemos usar funciones para conocer esas partes
- ¿Pero cómo definimos estos datos?
  - Hace falta una nueva **herramienta del lenguaje**

La expresión...

...describe el valor

número de ancho de espadas



1

palo de ancho de espadas



Espadas ▼

- Un registro es un caso de ***dato con estructura***
  - El **tipo** indica cuáles son los nombres de sus partes
    - Estas partes se llaman ***campos*** (*fields*)
    - Solo se puede definir un tipo nuevo en texto (no en bloques)
  - El **dato** se define indicando los valores de sus campos



- Solo se pueden definir tipos nuevos en texto (por ahora NO en bloques). Se usa la palabra clave:
  - **type** para un *tipo nuevo* (cuyo nombre va con mayúsculas)
  - **record** para un *registro*
  - **field** para cada *campo* (cuyo nombre va con minúsculas)
  - La elección de nombres sigue las reglas de siempre

```
type Carta is record {  
    field palo  
    field número  
}
```

Un valor de este tipo es un *registro* que tiene dos *campos*



- Solo se pueden definir tipos nuevos en texto (por ahora NO en bloques). Se usa la palabra clave:
  - **type** para un *tipo nuevo* (cuyo nombre va con mayúsculas)
  - **record** para un *registro*
  - **field** para cada *campo* (cuyo nombre va con minúsculas)
  - La elección de nombres sigue las reglas de siempre

```
type CartaEspañola is record {  
    field paloEspañol  
    field valorDeCartaEspañola  
}
```

Los nombres  
podrían ser  
otros

- Al definir un tipo registro, se debe dejar claro cuál es el tipo de datos que modela
- Para esto se debe escribir el ***propósito*** del tipo
  - Forma parte del *contrato* de la definición


```
type Carta is record {  
  /* PROPÓSITO: modelar cartas españolas de Truco  
  */  
  field palo  
  field número  
}
```

El propósito de un tipo usualmente es ***modelar*** un dato



- Un campo de registro puede tomar valores de cualquier tipo de datos
- Sin embargo, se espera que se utilice siempre el mismo campo con valores del mismo tipo
  - Se debe agregar como parte del *contrato* del tipo

```
type Carta is record {  
  /* PROPÓSITO: modelar cartas españolas de Truco */  
  field palo      // Un valor de tipo Palo  
  field número    // Un valor de tipo Número  
}
```



Si son cartas, no tiene sentido que el número sea un color...



- Para construir valores del tipo
  - Se usa el nombre del tipo como **constructor**
  - Se da valor a los campos usando el símbolo `<-` para cada nombre de campo (el orden no importa)
    - $\langle \text{NombreTipo} \rangle (\langle \text{campo}_1 \rangle \leftarrow \langle \text{exp}_1 \rangle$   
 $, \dots$   
 $, \langle \text{campo}_N \rangle \leftarrow \langle \text{exp}_N \rangle)$

El valor de esta expresión es la carta ancho de espadas

```
Carta(palo    <- Espadas  
      , número <- 1  
      )
```

la carta 1 de Espadas



- Cualquier combinación de valores es posible
  - Pero no todas se consideran adecuadas
  - ¿Cómo saber si un valor es válido en el tipo?

```
Carta(palo <- Bastos, número <- -10)
```

```
Carta(palo <- Espadas, número <- 27)
```

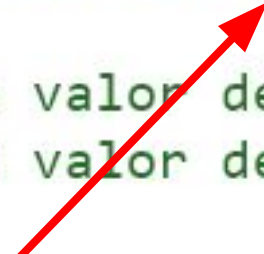
Son valores válidos,  
pero NO SON  
*verdaderas CARTAS*

```
Carta(palo <- Norte, número <- Rojo)
```

**Los valores de los campos deben  
cumplir ciertas condiciones**

- Las condiciones necesarias se dan en la definición
  - A estas condiciones las llamamos ***invariante de representación***
  - Es como “la precondition de los datos”

```
type Carta is record {  
  /* PROPÓSITO: modelar cartas españolas de Truco  
  INV.REP.:  
    - el número está entre 1 y 7, o entre 10 y 12  
  */  
  field palo      // Un valor de tipo Palo  
  field número   // Un valor de tipo Número  
}
```



Son cartas españolas de 40 naipes  
(sin 8s, ni 9s, ni comodines)



- Si se arma un dato que no cumple el invariante de representación, se considera *inválido*
  - Sin embargo, el lenguaje lo aceptará
  - Es responsabilidad del programador respetar los invariantes

```
Carta(palo    <- Espadas  
      , número <- 42  
      )
```

Este NO es un valor válido del tipo Carta  
(NO EXISTE el 42 de Espadas)



- Los registros se pueden
  - pasar como argumento de operaciones
  - recordar en variables
  - devolver como resultado de funciones
- ¡Son datos!

```
envidoSimpleCon_Y_(Carta(palo    <- Espadas  
                        , número <- 1)  
                  , Carta(palo    <- Espadas  
                        , número <- 7))
```

```
cartaLeída := cartaActual()
```



- Considerar la definición del tipo `Celda`

```
type Celda is record {  
    /*  
        PROPÓSITO: modelar una celda del tablero  
        INV.REP.: los números son todos  $\geq 0$   
    */  
    field cantidadDeAzules    // Un Número  
    field cantidadDeNegras    // Un Número  
    field cantidadDeRojas     // Un Número  
    field cantidadDeVerdes    // Un Número  
}
```



- Escribir una función `celdaActual` que describa la representación de la celda actual como valor del tipo `Celda` recién definido





- Escribir una función `celdaActual` que describa la representación de la celda actual como valor del tipo `Celda` recién definido
- SOLUCIÓN:

```
function celdaActual() {  
  /* PROPÓSITO: describe una celda del tablero como un  
    registro de tipo Celda  
    PRECONDICIONES: ninguna  
    RESULTADO: un valor de tipo Celda  
  */  
  return(Celda(cantidadDeAzules <- nroBolitas(Azul)  
    ,cantidadDeNegras <- nroBolitas(Negro)  
    ,cantidadDeRojas <- nroBolitas(Rojo)  
    ,cantidadDeVerdes <- nroBolitas(Verde)))  
}
```



- Cada campo tiene asociada una función de acceso llamada ***observador de campo*** o ***función observadora***
  - El nombre de la función es el mismo nombre del campo
  - Su argumento es un valor del tipo registro correspondiente
  - Describe el valor del campo dado

```
function enviadoSimpleCon_Y_(carta1, carta2) {  
  /* PROPÓSITO: describe el valor del enviado simple de  
               las 2 cartas dadas  
   PRECONDICIÓN: las cartas son del mismo palo y no son figuras  
   PARÁMETROS: ambos parámetros son de tipo Carta  
   RESULTADO: un Número, el valor del enviado  
  */  
  return (número(carta1) + número(carta2) + 20)  
}
```

Describen el número de cada carta



- Escribir una función `la_y_SonDelMismoPalo`, que dadas 2 cartas, indique si ambas son del mismo palo





- Escribir una función `la_y_SonDelMismoPalo`, que dadas 2 cartas, indique si ambas son del mismo palo
- SOLUCIÓN:

```
function la_y_SonDelMismoPalo(carta1, carta2) {  
  /* PROPÓSITO: indica si las cartas dadas son del mismo palo  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: ambos son de tipo Carta  
    RESULTADO: un Booleano, Verdadero si tienen el mismo  
                palo y Falso si no  
  */  
  return (palo(carta1) == palo(carta2))  
}
```

¡Comparamos el resultado de los observadores del campo palo!



# Variantes



- ¿Cómo modelar el palo de una carta?
  - No es un registro, porque no tiene partes
  - Hay 4 palos distintos
  - Es necesaria una nueva **herramienta del lenguaje**



No son  
Números, ni  
Colores, ni  
Direcciones

- En un ***tipo variante*** los valores son de distinta forma
  - Se define indicando los casos de la variación
  - Por ahora solo en texto (NO en bloques)
  - Se usan las palabras clave
    - **variant** para indicar que es un variante
    - **case** para indicar cada uno de los casos

```
type Palo is variant {  
    /* PROPÓSITO: modelar los palos  
    de cartas españolas  
    */  
    case Bastos    {}  
    case Copas     {}  
    case Espadas   {}  
    case Oros      {}  
}
```

Este tipo admite 4  
valores posibles



- Cada valor de un tipo variante se define con uno de los constructores de casos
  - Cada caso define un *constructor*
  - Los constructores enumeran los valores posibles
  - Por eso se conocen también como ***tipos enumerativos***

Estos son los 4  
posibles valores  
del tipo Palo



Copas  
Oros  
Espadas  
Bastos

- Los valores de un tipo *enumerativo* se pueden usar como cualquier otro valor
  - Como argumentos, en variables, o en campos
  - ¿Conocen ya algún tipo enumerativo predefinido?

```
function anchoDe_(paloAUsar){  
  /*  
    PROPÓSITO: describe el ancho del palo dado  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: paloAUsar es de tipo Palo  
    RESULTADO: una Carta, el ancho correspondiente  
  */  
  return (Carta(palo <- paloAUsar  
                , número <- 1))  
}
```

Parámetro de tipo Palo

Campo de tipo Palo



- Para decidir qué devolver según qué valor es
  - Se puede usar un choose con igualdades
  - O se puede usar otra herramienta (que no veremos en esta materia)

**Ejemplo:** Codificar palos con números



→ 100



→ 300



→ 200



→ 400



- Para decidir qué devolver según qué valor es
  - Se puede usar un choose con igualdades

```
function códigoDelPalo_(palo) {  
  /*  
    PROPÓSITO: describe el código del palo dado  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: palo es de tipo Palo  
    RESULTADO: un Número, el código correspondiente  
  */  
  return(choose 100 when (palo==Bastos)  
            200 when (palo==Copas)  
            300 when (palo==Espadas)  
            400 when (palo==Oros)  
            boom("Esto NO debería pasar. ¿Alteraste los palos?")  
            otherwise  
  )  
}
```

Describe solo una de las alternativas, según el parámetro



- Definir una función `siguientePalo_` que tome un palo y describa al palo siguiente al dado en el orden alfabético (circularmente, como en los colores)



- Definir una función `siguientePalo_` que tome un palo y describa al palo siguiente al dado en el orden alfabético (circularmente, como en los colores)
- SOLUCIÓN:

```
function siguientePalo_(palo) {  
  /* PROPÓSITO: describe el palo siguiente al dado  
    (en orden alfabético)  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: palo es de tipo Palo  
    RESULTADO: un Palo, el siguiente al dado  
  */  
  return(choose  
    Copas    when (palo==Bastos)  
    Espadas  when (palo==Copas)  
    Oros     when (palo==Espadas)  
    Bastos   when (palo==Oros)  
    boom("Esto NO debería pasar. ¿Alteraste los palos?")  
    otherwise  
  )  
}
```

Describe solo una de las alternativas, según el parámetro


- Usando `siguientePalo_` se puede hacer un recorrido sobre los palos
  - Ejemplo: poner los 4 anchos en el tablero

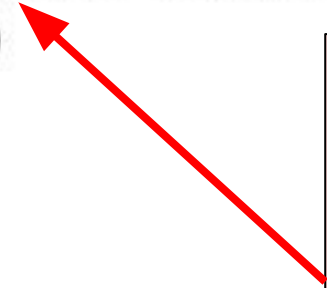
```
procedure PonerLosAnchos() {  
  /* PROPÓSITO: poner los 4 anchos en el tablero  
    PRECONDICIONES: hay 3 celdas al Este de la actual  
    OBSERVACIÓN: es un recorrido sobre los palos  
  */  
  paloActual := Bastos  
  while (paloActual /= Oros) {  
    PonerCarta_(anchoDe_(paloActual))  
    Mover(Este)  
    paloActual := siguientePalo_(paloActual)  
  }  
  PonerCarta_(anchoDe_(paloActual))  
}
```



## Más sobre registros

- Se puede construir un registro basándose en otro registro dado
  - Se puede hacer campo a campo
  - Se puede usar una notación especial

```
function la_ConvertidaAOros(cartaAnterior) {  
  /*  */  
  return(Carta(número <- número(cartaAnterior)  
              ,palo   <- Oros))  
}
```





Los campos  
que no  
cambian se  
copian  
usando los  
observadores





- Se puede construir un registro basándose en otro registro dado
  - Se puede hacer campo a campo
  - Se puede usar una notación especial

```
//function la_ConvertidaAOros(cartaAnterior) {  
//  /**/  
//  return(Carta(número <- número(cartaAnterior)  
//             ,palo   <- Oros))  
//}
```


```
function la_ConvertidaAOros(cartaAnterior) {  
  /**/  
  return(Carta(cartaAnterior | palo <- Oros))  
}
```

Solo se indican los que cambian respecto del valor anterior



- Se puede construir un registro basándose en otro registro dado

- $\langle \text{NombreTipo} \rangle (\langle \text{expresiónDeRegistro} \rangle \mid$   
     $\langle \text{campo}_1 \rangle \leftarrow \langle \text{exp}_1 \rangle$   
    ,  
     $\dots$   
    , $\langle \text{campo}_N \rangle \leftarrow \langle \text{exp}_N \rangle)$

```
function la_ConvertidaAOros(cartaAnterior) {  
  /*  */  
  return(Carta(cartaAnterior | palo <- Oros))  
}
```



valor nuevo



valor anterior




cambios

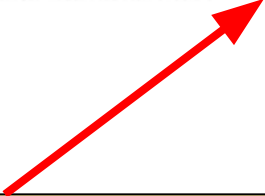


- Escribir una función `la_SinLasRojas` que dada una Celda (como registro de tipo Celda), describa la Celda resultante al sacar todas las bolitas rojas de la dada



- Escribir una función `la_SinLasRojas` que dada una Celda (como registro de tipo Celda), describa la Celda resultante al sacar todas las bolitas rojas de la dada
- SOLUCIÓN:

```
function la_SinLasRojas(celdaAnterior) {  
  /*  */  
  return(Celda(celdaAnterior | cantidadDeRojas <- 0))  
}
```




Las otras cantidades  
no se modifican



- Escribir una función **1a\_Con10AzulesMás** que dada una Celda (como registro de tipo Celda), describa la Celda resultante de agregar 10 bolitas azules a la dada



- Escribir una función **la\_Con10AzulesMás** que dada una Celda (como registro de tipo Celda), describa la Celda resultante de agregar 10 bolitas azules a la dada
- SOLUCIÓN:

```
function la_Con10AzulesMás(celdaAnterior) {  
  /*  */  
  return(Celda(celdaAnterior |  
    cantidadDeAzules <-  
      cantidadDeAzules(celdaAnterior) + 10))  
}
```

Las otras cantidades  
no se modifican

El valor nuevo es  
el anterior + 10



- Se puede usar un registro como valor del campo de otro
- Veamos un ejemplo: jugadores y equipos

```
type Jugador is record {  
  /* PROPÓSITO: modelar un jugador  
   INV.REP.: nombre no puede estar vacío  
   iniciativa está entre 0 y 100  
   fuerza es >= 0  
 */  
  field nombre      // String  
  field vida        // Número  
  field fuerza      // Número  
  field ataqueBásico // Ataque  
  field iniciativa  // Número  
}
```

```
type Ataque is variant {  
  /* PROPÓSITO: modelar los tipos de ataque  
  */  
  case Puñetazo {}  
  case Patada {}  
  case Mordisco {}  
  case Cabezazo {}  
  case Rodillazo {}  
}
```

La única novedad hasta acá es un tipo básico nuevo



- Se puede usar un registro como valor del campo de otro
- Veamos un ejemplo: jugadores y equipos

```
type Equipo is record {  
  /* PROPÓSITO: modelar un equipo de juego  
    INV.REP.: deben ser 3 jugadores diferentes  
  */  
  field jugadorIzquierdo // Jugador  
  field jugadorCentro     // Jugador  
  field jugadorDerecho    // Jugador  
}
```

¡Un jugador es un registro!

Un Equipo es un registro con 3 registros de Jugador



- Se puede usar un registro como valor del campo de otro
- Veamos un ejemplo: jugadores y equipos

3 jugadores

```
function manchú() {  
  // PROPÓSITO: describe al jugador Manchú  
  return(Jugador(nombre    <- "Manchú"  
                  , vida    <- 300  
                  , fuerza   <- 20  
                  , ataqueBasico <- Cabezazo  
                  , iniciativa <- 30))  
}
```

Los **Strings** son  
cadenas de caracteres  
entre comillas dobles

```
function toto() {  
  // PROPÓSITO: describe al jugador Toto  
  return(Jugador(nombre    <- "Toto"  
                  , vida    <- 200  
                  , fuerza   <- 30  
                  , ataqueBasico <- Patada  
                  , iniciativa <- 25))  
}
```


```
function serena() {  
  // PROPÓSITO: describe a la jugadora Serena  
  return(Jugador(nombre    <- "Serena"  
                  , vida    <- 400  
                  , fuerza   <- 18  
                  , ataqueBasico <- Mordisco  
                  , iniciativa <- 55))  
}
```





- Se puede usar un registro como valor del campo de otro
- Veamos un ejemplo: jugadores y equipos


```
function elEquipoDeToto() {  
  // PROPÓSITO: describe el equipo de Toto  
  return (Equipo(jugadorIzquierdo <- manchú()  
                 , jugadorCentro    <- serena()  
                 , jugadorDerecho   <- toto()))  
}
```



Los 3 jugadores son valores de campo en el Equipo

- Se puede usar un registro como valor del campo de otro
- Veamos un ejemplo: jugadores y equipos

```
function nombreDelCapitán_(equipo) {  
  /* PROPÓSITO: describe el nombre del  
                capitán del equipo dado  
  PRECONDICIÓN: ninguna  
  PARÁMETRO: equipo es de tipo Equipo  
  RESULTADO: un String, el nombre pedido  
  */  
  return (nombre(jugadorDerecho(equipo)))  
}
```



¡El jugador derecho es un  
registro con campo nombre!



- Se puede usar un registro como valor del campo de otro
- Veamos un ejemplo: jugadores y equipos

```
function el_ConPunterosIntercambiados(equipo) {  
  /* PROPÓSITO: describe un equipo como el dado, pero  
    con los punteros intercambiados  
    PRECONDICIÓN: ninguna  
    PARÁMETRO: equipo es de tipo Equipo  
    RESULTADO: un Equipo  
  */  
  return (Equipo(equipo  
    | jugadorIzquierdo <- jugadorDerecho(equipo)  
    , jugadorDerecho   <- jugadorIzquierdo(equipo)))  
}
```

Hacer un registro de registros basado en otro sigue las mismas reglas de siempre



# Cierre

- **Registros**

- Son datos con estructura
- Su estructura está formada por **campos**
  - Cada campo tiene su nombre y un tipo
- Cada campo define una función observadora
- Como son datos, se pueden usar donde se esperan datos (parámetros, variables, resultados, campos)
- Hay una notación para acortar la creación de registros basados en otros
- El tipo tiene un **contrato** que hay que establecer
  - Propósito e invariante de representación
  - Tipos de los campos



- ***Variantes***

- Son datos con estructura
- Su estructura está dada por ***casos***
- Cada caso tiene un nombre y define un valor diferente del mismo tipo
- Los ***tipos enumerativos*** son un ejemplo de variante