



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 PonerCartaFEO(númeroDeLaCarta, paloDeLaCarta) {  
  /* PROPÓSITO: pone la codificación de bolitas de una carta  
  PRECONDICIÓN:  
    * el número está entre 1 y 7, o entre 10 y 12  
    * el palo de la carta es uno de los 4 válidos  
  PARÁMETROS:  
    * númeroDeLaCarta es un número  
    * paloDeLaCarta es un palo (como sea que se represente)  
  */  
  PonerMuchas(3, Azul)  
  PonerMuchas(códigoDeCarta(númeroDeLaCarta, paloDeLaCarta), Negro)  
}
```

Dos
parámetros,
muchas
precondiciones

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

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



¿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



- 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





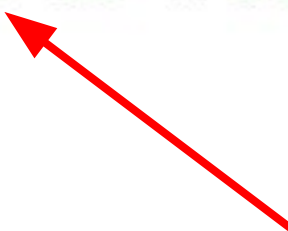
- Solo se pueden definir tipos nuevos en texto (por ahora NO en bloques). Se usa la palabra clave:
 - **type** para un *tipo nuevo* (cuyo nombre va con mayúsculas)
 - **record** para un *registro*
 - **field** para cada *campo* (cuyo nombre va con minúsculas)
 - La elección de nombres sigue las reglas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

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

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

- 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)



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

```
envidoSimple(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 >= 0  
    */  
    field cantidadDeAzules    // Un Número  
    field cantidadDeNegras    // Un Número  
    field cantidadDeRojas     // Un Número  
    field cantidadDeVerdes    // Un Número  
}
```



- Considerar la definición del tipo **Celda**
- Escribir una función **celdaActual()** que retorne la representación de la celda actual como valor del tipo





- Considerar la definición del tipo **Celda**
- Escribir una función **celdaActual()** que retorne la representación de la celda actual como valor del tipo
- SOLUCIÓN:

```
function celdaActual() {  
  /*  
    PROPÓSITO: describe una celda del tablero como  
               un registro de tipo Celda  
    PRECONDICIÓN: ninguna  
  */  
  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
 - Retorna el valor del campo dado

```
function envideoSimple(carta1, carta2) {  
    /* PROPÓSITO: calcula el envideo simple de 2 cartas  
       PRECONDICIÓN: las cartas son del mismo palo  
                      y no son figuras  
       PARÁMETROS: ambos parámetros son de tipo carta  
    */  
    return (número(carta1) + número(carta2) + 20)  
}
```



Describen el número de cada carta



- Escribir una función **sonDelMismoPalo** que dadas 2 cartas, describa Verdadero si ambas son del mismo palo



- Escribir una función `sonDelMismoPalo` que dadas 2 cartas, describa Verdadero si ambas son del mismo palo
- SOLUCIÓN:

```
function sonDelMismoPalo(carta1, carta2) {  
    /* PROPÓSITO: describir Verdadero si ambas  
               cartas son del mismo palo  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: ambos son de tipo carta  
    */  
    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
```

```
  */
```

```
  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 una nueva herramienta:
 - ***Alternativa indexada en expresiones***

Ejemplo: Codificar palos con números



→ 100



→ 300



→ 200



→ 400

- La **alternativa indexada en expresiones** se define en texto usando la palabra reservada **matching**
 - **matching** (<expVariante>) **select**
 <exp₁> **on** <caso₁> ... <exp_N> **on** <caso_N>
 <exp> **otherwise**

```
function códigoDelPalo(palo) {  
  /*  
    PROPÓSITO: describe el código del palo  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: palo es de tipo Palo  
  */  
  return(matching (palo) select  
    100 on Bastos  
    200 on Copas  
    300 on Espadas  
    400 on Oros  
    boom("¿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 (en orden alfabético)  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: palo es de tipo Palo  
  */  
  return(matching (palo) select  
    Copas    on Bastos  
    Espadas  on Copas  
    Oros     on Espadas  
    Bastos   on Oros  
    boom("¿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  
       PRECONDICIÓN: 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))  
}
```

- La **alternativa indexada en comandos** se define en texto usando la palabra reservada **switch**

```
○ switch(<expVariante>) {  
    <casoI> -> { <comandosI> }  
    ...  
    <casoN> -> { <comandosN> }  
    _      -> { <comandos> }  
}
```


```
procedure IndicarPaloPorColor(palo) { /*  */  
    switch (palo) {  
        Bastos    -> { Poner(Verde) }  
        Copas     -> { Poner(Rojo)  }  
        Espadas   -> { Poner(Azul)   }  
        Oros      -> { Poner(Negro)  }  
        _ -> { BOOM("¿Alteraste los palos?") }  
    }  
}
```

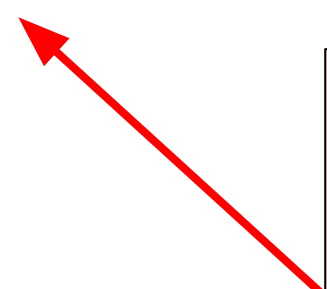
Solamente pone
una bolita



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

```
function 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

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

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

valor nuevo

valor anterior


cambios

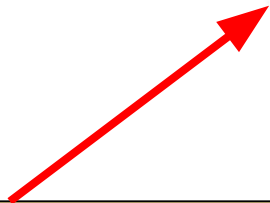


- Escribir una función **sinLasRojas** que dada una Celda (como registro de tipo Celda), describa la Celda resultante al sacar todas las bolitas rojas de la dada



- Escribir una función **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 sinLasRojas(celdaAnterior) {  
  /*  */  
  return(Celda(celdaAnterior | cantidadDeRojas <- 0))  
}
```



Las otras cantidades
no se modifican




- Escribir una función **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 **con10AzulesMás** que dada una Celda (como registro de tipo Celda), describa la Celda resultante de agregar 10 bolitas azules a la dada
- SOLUCIÓN:

```
function 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 ataque      // 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()))  
}
```

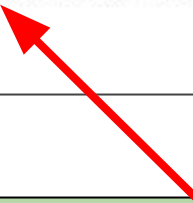


Los 3 jugadores son valores de campo en el Equipo



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

```
function nombreDelCapitán(equipo) {  
  /* PROPÓSITO: retornar el nombre del capitán  
    PRECONDICIÓN: ninguna  
    PARÁMETRO: equipo es de tipo Equipo  
  */  
  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 punterosIntercambiados(equipo) {  
  /* PROPÓSITO: retornar el equipo con sus punteros intercambiados  
  PRECONDICIÓN: ninguna  
  PARÁMETRO: equipo es de tipo 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, 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
- Se pueden inspeccionar usando ***alternativa indexada*** (en expresiones y en comandos)