Un recorrido por **Conceptos** generales y específicos de Programación

Introducción

Juan Francisco Díaz Frias

Maestría en Ingeniería, Énfasis en Ingeniería de Sistemas y Computación
Escuela de Ingeniería de Sistemas y Computación,
home page: http://eisc.univalle.edu.co
Universidad del Valle - Cali, Colombia

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Cálculos sencillos

Iniciar Oz

Inicie Mozart digitando `oz` o haciendo doble clic en el ícono de Mozart.

Lanzar un cálculo

```
{Browse 9999*9999}
```

Cálculos sencillos

Iniciar Oz

Inicie Mozart digitando `oz` o haciendo doble clic en el ícono de Mozart.

Lanzar un cálculo

```
{Browse 9999*9999}
```

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Variables declarativas

Declaración

```
declare
 V=9999*9999
{Browse V*V}
```

Propiedades

- Las variables son solamente abreviaciones para valores.
- Ellas no pueden ser asignadas más de una vez.
- Identificador de variable: Es lo que se digita. Ej: `Var1`
- La variable del almacén. Es lo que el sistema utiliza para calcular. Hace parte de la memoria del sistema, la cual llamamos almacén.

Variables declarativas

Declaración

```
declare
 V=9999*9999
{Browse V*V}
```

Propiedades

- Las variables son solamente abreviaciones para valores.
- Ellas no pueden ser asignadas más de una vez.
- Identificador de variable: Es lo que se digita. Ej: `Var1`
- La variable del almacén. Es lo que el sistema utiliza para calcular. Hace parte de la memoria del sistema, la cual llamamos almacén.

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Funciones

Calcular $n!$

$n! = 1 \times 2 \times \cdots \times (n-1) \times n.$

¿Cómo? usando las definición matemática de factorial:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \text{ si } n > 0 \end{aligned}$$

En Oz ...

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

Se crea la variable `Fact`.

Ensayar `{Browse {Fact 100}}`

Funciones

Calcular $n!$

$n! = 1 \times 2 \times \cdots \times (n-1) \times n.$

¿Cómo? usando las definición matemática de factorial:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \text{ si } n > 0 \end{aligned}$$

En Oz ...

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

Se crea la variable `Fact`.

Ensayar `{Browse {Fact 100}}`

Componiendo funciones

Combinaciones

¿Cuantas combinaciones que se pueden formar con k elementos tomados de un conjunto de n elementos?

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

En Oz . . . Abstracción Funcional

```
declare
fun {Comb N K}
   {Fact N} div ({Fact K}*{Fact N-K})
end
```

Componiendo funciones

Combinaciones

¿Cuantas combinaciones que se pueden formar con k elementos tomados de un conjunto de n elementos?

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

En Oz . . . Abstracción Funcional

```
declare
fun {Comb N K}
   {Fact N} div ({Fact K}*{Fact N-K})
end
```

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Datos estructurados (listas) (1)

Problema: Calcular una fila del triángulo de Pascal

Qué estructura usar: listas

- `[5 6 7 8]`, `nil` (y no `[]`), `H|T`

```
declare
H=5
T=[6 7 8]
{Browse H|T}
```

Datos estructurados (listas) (1)

Problema: Calcular una fila del triángulo de Pascal

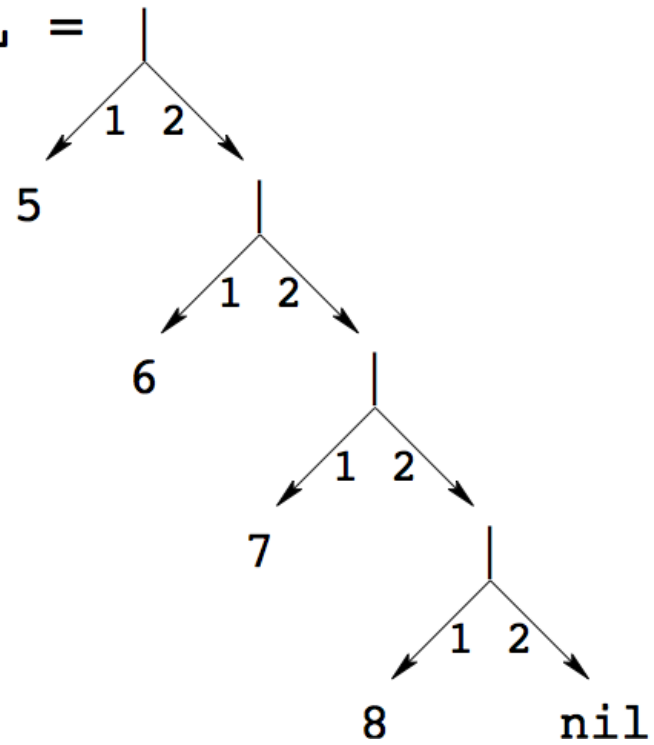
Qué estructura usar: listas

- `[5 6 7 8]`, `nil` (y no `[]`), `H|T`
-
```
declare
H=5
T=[6 7 8]
{Browse H|T}
```

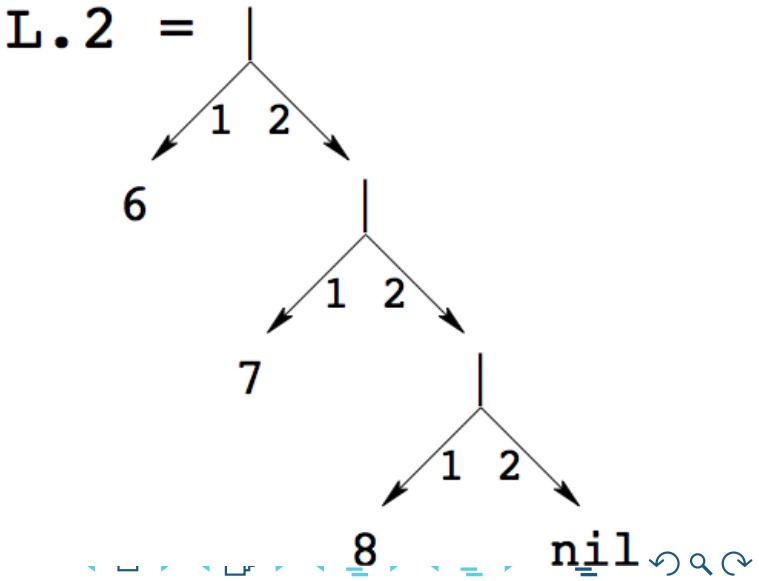
Datos estructurados (listas) (2)

Estructura de una lista

```
L = [5 6 7 8]
```




```
L.1 = 5
L.2 = [6 7 8]
```



Datos estructurados (listas) (3)

Reconocimiento de patrones: instrucción `case`

```
declare
L=[5 6 7 8]
case L of H|T then {Browse H} {Browse T} end
```

- `H` y `T` son variables locales,
- `case` descompone `L` de acuerdo al patrón `H|T`.

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

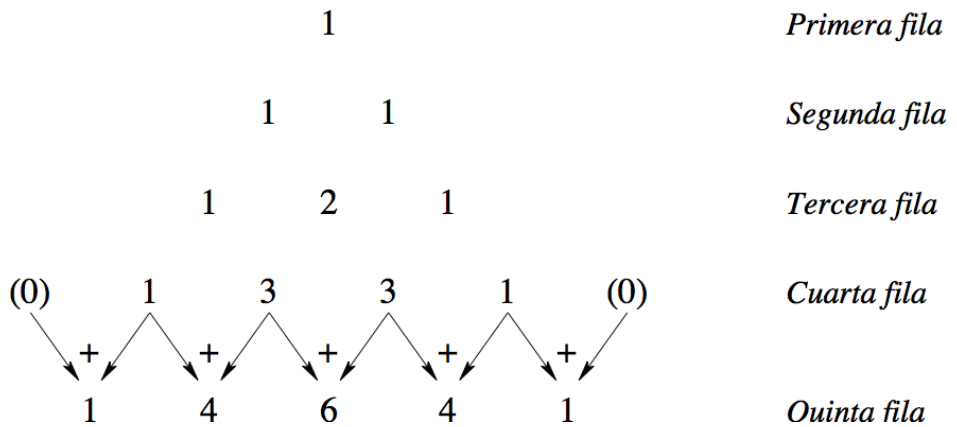
Funciones sobre listas (1)

Calcular `{Pascal N}`

- La fila `N` depende de la anterior
- Por ejemplo, tomemos la 4a. fila: `[1 3 3 1]`

  La 5a,

  `[1 4 6 4 1]` =
  `[1 3 3 1 0]` +
  `[0 1 3 3 1]`



Funciones sobre listas (2)

La función principal

```
declare Pascal SumListas DesplIzq
DesplDer
fun {Pascal N}
   if N==1 then [1]
   else
     {SumListas
       {DesplIzq {Pascal N-1}}
       {DesplDer {Pascal N-1}}}
   end
end
```

Las funciones auxiliares `DesplIzq` y `DesplDer`

```
fun {DesplIzq L}
   case L of H|T then
      H|{DesplIzq T}
   else [0] end
end

fun {DesplDer L} 0|L end
```

Funciones sobre listas (2)

La función principal

```
declare Pascal SumListas DesplIzq
DesplDer
fun {Pascal N}
   if N==1 then [1]
   else
     {SumListas
       {DesplIzq {Pascal N-1}}
       {DesplDer {Pascal N-1}}}
   end
end
```

Las funciones auxiliares `DesplIzq` y `DesplDer`

```
fun {DesplIzq L}
   case L of H|T then
      H|{DesplIzq T}
   else [0] end
end

fun {DesplDer L} 0|L end
```

Funciones sobre listas (3)

La función auxiliar `SumListas`

```
fun {SumListas L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         H1+H2|{SumListas T1 T2}
      end
   else nil end
end
```

Pruebas

Invocar `{Pascal 20}`:

```
[1 19 171 969 3876 11628 27132
50388 75582 92378
 92378 75582 50388 27132 11628
3876 969 171 19 1]
```

¿Es correcto?

Ensaye `{Pascal 24}`, `{Pascal 30}`, qué puede decir?

Funciones sobre listas (3)

La función auxiliar `SumListas`

```
fun {SumListas L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         H1+H2|{SumListas T1 T2}
      end
   else nil end
end
```

Pruebas

Invocar `{Pascal 20}`:

```
 [1 19 171 969 3876 11628 27132
50388 75582 92378
  92378 75582 50388 27132 11628
3876 969 171 19 1]
```

¿Es correcto?

Ensaye `{Pascal 24}`, `{Pascal 30}`, qué puede decir?

Universidad del Valle

Plan

Evaluación perezosa(1)

Evaluación ansiosa vs. perezosa

- Las funciones que hemos escrito hasta ahora realizan sus cálculos tan pronto como son invocadas.
- En la evaluación perezosa, solo se realiza un cálculo cuando su resultado se necesita.
- Uso: Representación de estructuras de datos infinitas.

```
fun lazy {Ents N}
   N|{Ents N+1}
end
```

- Ensayar:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

Evaluación perezosa(1)

Evaluación ansiosa vs. perezosa

- Las funciones que hemos escrito hasta ahora realizan sus cálculos tan pronto como son invocadas.
- En la evaluación perezosa, solo se realiza un cálculo cuando su resultado se necesita.
- Uso: Representación de estructuras de datos infinitas.

```
fun lazy {Ents N}
   N|{Ents N+1}
end
```

- Ensayar:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

Evaluación perezosa(1)

Evaluación ansiosa vs. perezosa

- Las funciones que hemos escrito hasta ahora realizan sus cálculos tan pronto como son invocadas.
- En la evaluación perezosa, solo se realiza un cálculo cuando su resultado se necesita.
- Uso: Representación de estructuras de datos infinitas.

```
fun lazy {Ents N}
   N|{Ents N+1}
end
```

- Ensayar:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

Evaluación perezosa(1)

Evaluación ansiosa vs. perezosa

- Las funciones que hemos escrito hasta ahora realizan sus cálculos tan pronto como son invocadas.
- En la evaluación perezosa, solo se realiza un cálculo cuando su resultado se necesita.
- Uso: Representación de estructuras de datos infinitas.

```
fun lazy {Ents N}
   N|{Ents N+1}
end
```

- Ensayar:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

Evaluación perezosa(2)

Cálculo perezoso del triángulo de Pascal

- Sencillo:

```
fun lazy {ListaPascal Fila}
   Fila|{ListaPascal
         {SumListas {DesplIzq Fila}
                    {DesplDer Fila}}}
end
```

- Calculemos:

```
declare
L={ListaPascal [1]}  {Browse L}
```

- Miremos diferentes filas:

```
{Browse L.1} {Browse L.2.1} {Browse L.2.2.2.2.1}
```

Evaluación perezosa(2)

Cálculo perezoso del triángulo de Pascal

- Sencillo:

```
fun lazy {ListaPascal Fila}
   Fila|{ListaPascal
         {SumListas {DesplIzq Fila}
                    {DesplDer Fila}}}
end
```

- Calculemos:

```
declare
L={ListaPascal [1]}  {Browse L}
```

- Miremos diferentes filas:

```
{Browse L.1} {Browse L.2.1} {Browse L.2.2.2.2.1}
```

Evaluación perezosa(2)

Cálculo perezoso del triángulo de Pascal

- Sencillo:

```
fun lazy {ListaPascal Fila}
   Fila|{ListaPascal
         {SumListas {DesplIzq Fila}
                    {DesplDer Fila}}}
end
```

- Calculemos:

```
declare
L={ListaPascal [1]}  {Browse L}
```

- Miremos diferentes filas:

```
{Browse L.1} {Browse L.2.1} {Browse L.2.2.2.2.1}
```

Evaluación perezosa(3)

Alternativa ansiosa

```
fun {ListaPascal2 N Fila}
   if N==1 then [Fila]
   else
      Fila|{ListaPascal2 N-1
            {SumListas {DesplIzq Fila}
                       {DesplDer Fila}}}
   end
end
{Browse {ListaPascal2 10 [1]}}
```

¿Y si necesitamos la fila 11?

Plan

Programación de alto orden (1)

Variaciones del triángulo de Pascal

- Suponga que en lugar de sumar los números de las listas para calcular cada fila, nos gustaría restarlos, o aplicarles el o-exclusivo, o muchas otras alternativas.
- Se puede escribir una nueva versión de `PascalRápido` para cada variación.
- O escribir una versión genérica a la que le pasamos la función adecuada a la variación como argumento.
- La capacidad de pasar funciones como argumentos se conoce como programación de alto orden.
- La programación de alto orden ayuda a construir abstracciones genéricas.

Programación de alto orden (1)

Variaciones del triángulo de Pascal

- Suponga que en lugar de sumar los números de las listas para calcular cada fila, nos gustaría restarlos, o aplicarles el o-exclusivo, o muchas otras alternativas.
- Se puede escribir una nueva versión de `PascalRápido` para cada variación.
- O escribir una versión genérica a la que le pasamos la función adecuada a la variación como argumento.
- La capacidad de pasar funciones como argumentos se conoce como programación de alto orden.
- La programación de alto orden ayuda a construir abstracciones genéricas.

Programación de alto orden (1)

Variaciones del triángulo de Pascal

- Suponga que en lugar de sumar los números de las listas para calcular cada fila, nos gustaría restarlos, o aplicarles el o-exclusivo, o muchas otras alternativas.
- Se puede escribir una nueva versión de `PascalRápido` para cada variación.
- O escribir una versión genérica a la que le pasamos la función adecuada a la variación como argumento.
- La capacidad de pasar funciones como argumentos se conoce como programación de alto orden.
- La programación de alto orden ayuda a construir abstracciones genéricas.

Programación de alto orden (1)

Variaciones del triángulo de Pascal

- Suponga que en lugar de sumar los números de las listas para calcular cada fila, nos gustaría restarlos, o aplicarles el o-exclusivo, o muchas otras alternativas.
- Se puede escribir una nueva versión de `PascalRápido` para cada variación.
- O escribir una versión genérica a la que le pasamos la función adecuada a la variación como argumento.
- La capacidad de pasar funciones como argumentos se conoce como programación de alto orden.
- La programación de alto orden ayuda a construir abstracciones genéricas.

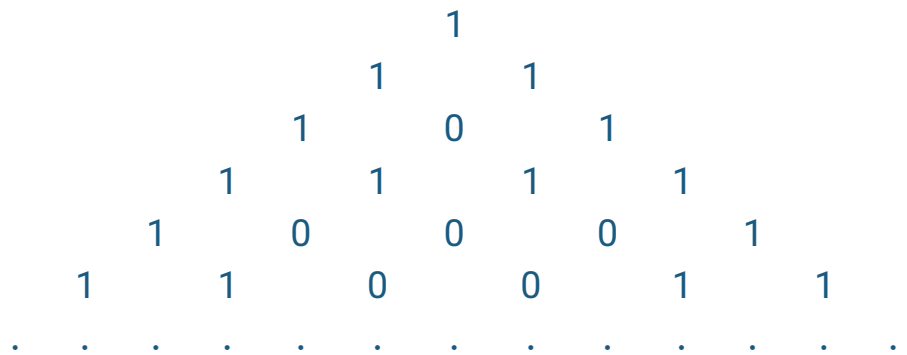
Programación de alto orden (1)

Variaciones del triángulo de Pascal

- Suponga que en lugar de sumar los números de las listas para calcular cada fila, nos gustaría restarlos, o aplicarles el o-exclusivo, o muchas otras alternativas.
- Se puede escribir una nueva versión de `PascalRápido` para cada variación.
- O escribir una versión genérica a la que le pasamos la función adecuada a la variación como argumento.
- La capacidad de pasar funciones como argumentos se conoce como programación de alto orden.
- La programación de alto orden ayuda a construir abstracciones genéricas.

Programación de alto orden (2)

Variaciones de `Pascal`



Programación de alto orden (3)

PascalGenérico

```
fun {PascalGenérico Op N}
   if N==1 then [1]
   else L in L={PascalGenérico Op N-1}
          {OpListas Op {DesplIzq L} {DesplDer L}}
   end
end
fun {OpListas Op L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         {Op H1 H2}|{OpListas Op T1 T2}
      end
   else nil end
end
```

Programación de alto orden (4)

Variaciones

```
fun {Sum X Y} X+Y end

fun {Xor X Y} if X==Y then 0 else 1 end end

fun {PascalRápido N} {PascalGenérico Sum N} end

fun {PascalXor N} {PascalGenérico Xor N} end

{Browse {PascalRápido 10}}

{Browse {PascalXor 10}}
```

Plan

Concurrencia

- Programar actividades independientes, cada una de las cuales se ejecuta a su propio ritmo.
- No debería existir interferencia entre la actividades, a menos que el programador decida que necesitan comunicarse.
- Un hilo es simplemente un programa secuencial en ejecución.
- Un programa puede tener más de un hilo. Los hilos se crean con la instrucción `thread`.
- `{Browse 99*99}` responde inmediatamente mientras se calcula `{Pascal 30}`

Ejemplo sencillo

```
thread P in
   P={Pascal 30}
   {Browse P}
end
{Browse 99*99}
```

Concurrencia

- Programar actividades independientes, cada una de las cuales se ejecuta a su propio ritmo.
- No debería existir interferencia entre la actividades, a menos que el programador decida que necesitan comunicarse.
- Un hilo es simplemente un programa secuencial en ejecución.
- Un programa puede tener más de un hilo. Los hilos se crean con la instrucción `thread`.
- `{Browse 99*99}` responde inmediatamente mientras se calcula `{Pascal 30}`

Ejemplo sencillo

```
thread P in
   P={Pascal 30}
   {Browse P}
end
{Browse 99*99}
```

Concurrencia

- Programar actividades independientes, cada una de las cuales se ejecuta a su propio ritmo.
- No debería existir interferencia entre la actividades, a menos que el programador decida que necesitan comunicarse.
- Un hilo es simplemente un programa secuencial en ejecución.
- Un programa puede tener más de un hilo. Los hilos se crean con la instrucción `thread`.
- `{Browse 99*99}` responde inmediatamente mientras se calcula `{Pascal 30}`

Ejemplo sencillo

```
thread P in
   P={Pascal 30}
   {Browse P}
end
{Browse 99*99}
```

Concurrencia

- Programar actividades independientes, cada una de las cuales se ejecuta a su propio ritmo.
- No debería existir interferencia entre la actividades, a menos que el programador decida que necesitan comunicarse.
- Un hilo es simplemente un programa secuencial en ejecución.
- Un programa puede tener más de un hilo. Los hilos se crean con la instrucción **`thread`**.
- `{Browse 99*99}` responde inmediatamente mientras se calcula `{Pascal 30}`

Ejemplo sencillo

```
thread P in
   P={Pascal 30}
   {Browse P}
end
{Browse 99*99}
```

Concurrencia

- Programar actividades independientes, cada una de las cuales se ejecuta a su propio ritmo.
- No debería existir interferencia entre la actividades, a menos que el programador decida que necesitan comunicarse.
- Un hilo es simplemente un programa secuencial en ejecución.
- Un programa puede tener más de un hilo. Los hilos se crean con la instrucción **`thread`**.
- `{Browse 99*99}` responde inmediatamente mientras se calcula `{Pascal 30}`

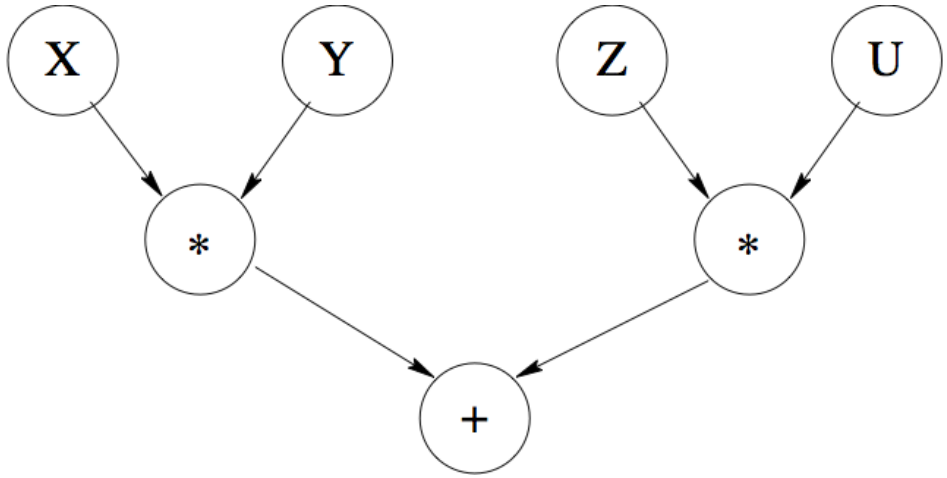
Ejemplo sencillo

```
thread P in
   P={Pascal 30}
   {Browse P}
end
{Browse 99*99}
```

Flujo de datos (1)

Variables de flujo de datos

- ¿Qué pasa cuando varios hilos tratan de comunicarse?
- Se puede hacer que se sincronicen sobre la disponibilidad de los datos (ejecución dirigida por los datos).
- Si una operación trata de usar una variable que no esté ligada aún, entonces espera.
- Esa variable es una variable de flujo de datos.

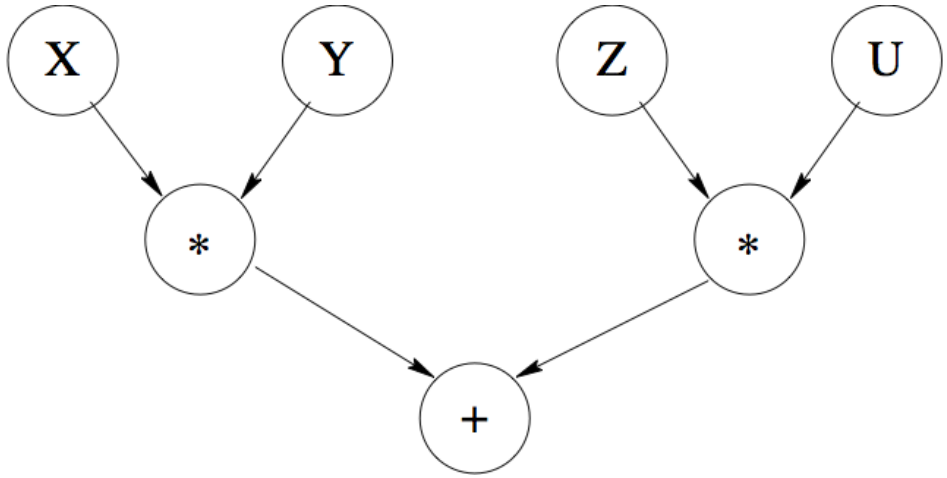


Flujo de datos (1)

Variables de flujo de datos

- ¿Qué pasa cuando varios hilos tratan de comunicarse?
- Se puede hacer que se sincronicen sobre la disponibilidad de los datos (ejecución dirigida por los datos).
- Si una operación trata de usar una variable que no esté ligada aún, entonces espera.
- Esa variable es una variable de flujo de datos.

Ejecución por flujo de datos

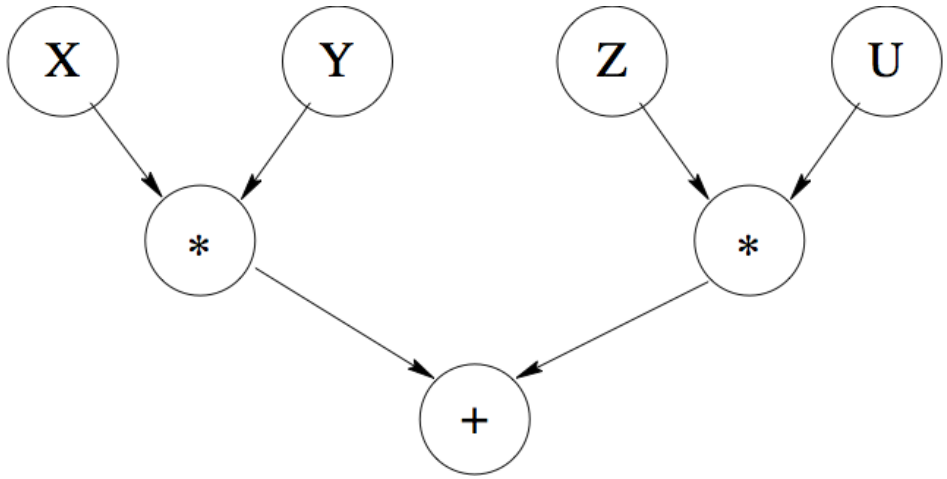


Flujo de datos (1)

Variables de flujo de datos

- ¿Qué pasa cuando varios hilos tratan de comunicarse?
- Se puede hacer que se sincronicen sobre la disponibilidad de los datos (ejecución dirigida por los datos).
- Si una operación trata de usar una variable que no esté ligada aún, entonces espera.
- Esa variable es una variable de flujo de datos.

Ejecución por flujo de datos

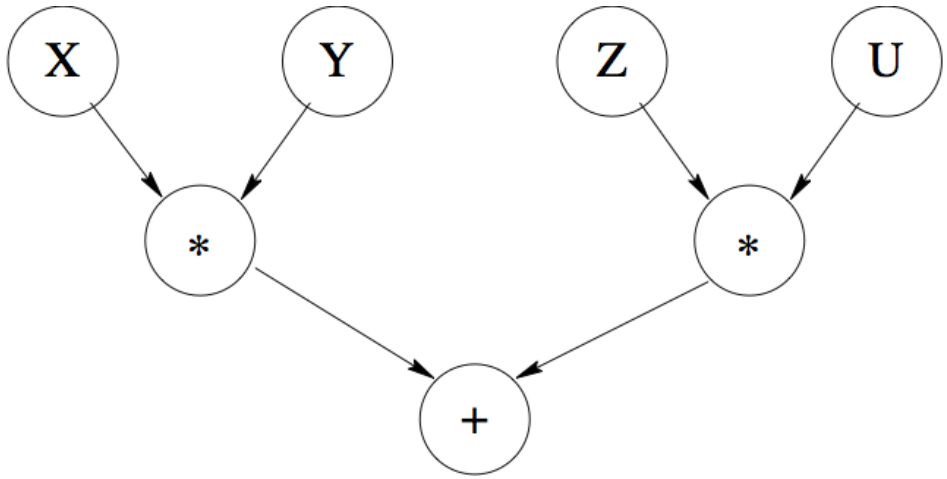


Flujo de datos (1)

Variables de flujo de datos

- ¿Qué pasa cuando varios hilos tratan de comunicarse?
- Se puede hacer que se sincronicen sobre la disponibilidad de los datos (ejecución dirigida por los datos).
- Si una operación trata de usar una variable que no esté ligada aún, entonces espera.
- Esa variable es una variable de flujo de datos.

Ejecución por flujo de datos

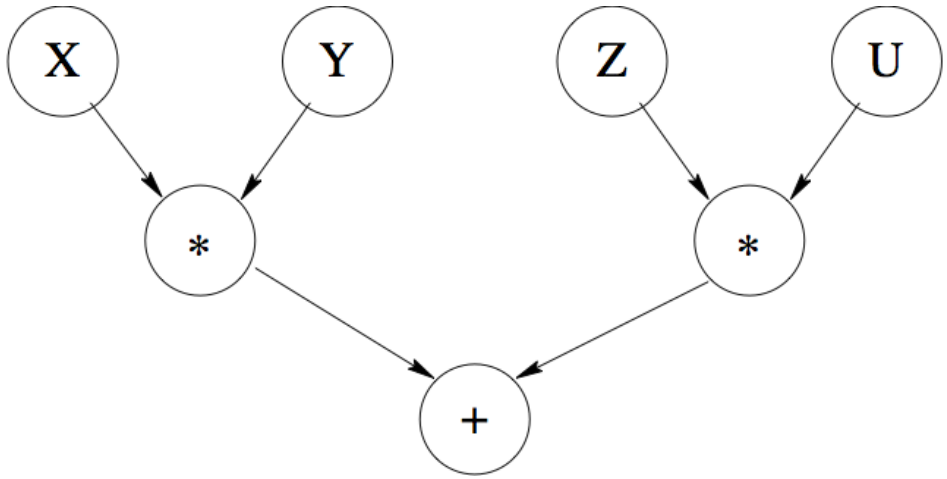


Flujo de datos (1)

Variables de flujo de datos

- ¿Qué pasa cuando varios hilos tratan de comunicarse?
- Se puede hacer que se sincronicen sobre la disponibilidad de los datos (ejecución dirigida por los datos).
- Si una operación trata de usar una variable que no esté ligada aún, entonces espera.
- Esa variable es una variable de flujo de datos.

Ejecución por flujo de datos



Flujo de datos (2)

Propiedades del flujo de datos

- Los cálculos funcionan correctamente independientemente de cómo se han repartido entre los hilos.
- Los cálculos son pacientes: no señalan errores, sino que simplemente esperan.

Propiedades del flujo de datos

Tienen sentido cuando los cálculos están compuestos por múltiples hilos.

Siempre el mismo resultado

```
declare X in
thread
{Delay 10000} X=99
end
{Browse ini} {Browse X*X}
```

vs

```
declare X in
thread
{Browse ini} {Browse X*X}
end
{Delay 10000} X=99
```

Flujo de datos (2)

Propiedades del flujo de datos

- Los cálculos funcionan correctamente independientemente de cómo se han repartido entre los hilos.
- Los cálculos son pacientes: no señalan errores, sino que simplemente esperan.

Propiedades del flujo de datos

Tienen sentido cuando los cálculos están compuestos por múltiples hilos.

Siempre el mismo resultado

```
declare X in
thread
{Delay 10000} X=99
end
{Browse ini} {Browse X*X}
```

vs

```
declare X in
thread
{Browse ini} {Browse X*X}
end
{Delay 10000} X=99
```

Universidad del Valle

Plan

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Celdas

Ejemplo Celdas

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Celdas

Ejemplo Celdas

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Celdas

Ejemplo Celdas

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Celdas

- Una celda es un contenedor de un valor.
- Se pueden crear celdas, acceder al valor que contiene o modificarlo.
- Las celdas no son variables.

Ejemplo Celdas

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Celdas

- Una celda es un contenedor de un valor.
- Se pueden crear celdas, acceder al valor que contiene o modificarlo.
- Las celdas no son variables.

Ejemplo Celdas

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Celdas

- Una celda es un contenedor de un valor.
- Se pueden crear celdas, acceder al valor que contiene o modificarlo.
- Las celdas no son variables.

Ejemplo Celdas

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Estado (1)

Estado explícito

- ¿Cómo podemos lograr que una función aprenda de su pasado?
- Nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea.
- La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado.
- Este tipo de memoria se llama estado explícito.El estado explícto modela un aspecto esencial de la manera como el mundo real funciona.
- Necesitamos un concepto para almacenar, modificar y recuperar un valor. El concepto más sencillo es una celda.

Celdas

- Una celda es un contenedor de un valor.
- Se pueden crear celdas, acceder al valor que contiene o modificarlo.
- Las celdas no son variables.

Ejemplo Celdas

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Estado (2)

Agregando memoria a `PascalRápido`

- Con una celda, podemos lograr que `PascalRápido` cuente cuántas veces es invocada.
- La memoria (estado) es global en este caso.
- La memoria local a una función se denomina estado encapsulado.

Pascal con memoria

```
declare
C={NewCell 0}
fun {PascalRápido N}
   C:=@C+1
   {PascalGenérico Sum N}
end
```

Estado (2)

Agregando memoria a `PascalRápido`

- Con una celda, podemos lograr que `PascalRápido` cuente cuántas veces es invocada.
- La memoria (estado) es global en este caso.
- La memoria local a una función se denomina estado encapsulado.

Pascal con memoria

```
declare
C={NewCell 0}
fun {PascalRápido N}
   C:=@C+1
   {PascalGenérico Sum N}
end
```

Estado (2)

Agregando memoria a `PascalRápido`

- Con una celda, podemos lograr que `PascalRápido` cuente cuántas veces es invocada.
- La memoria (estado) es global en este caso.
- La memoria local a una función se denomina estado encapsulado.

Pascal con memoria

```
declare
C={NewCell 0}
fun {PascalRápido N}
   C:=@C+1
   {PascalGenérico Sum N}
end
```

Estado (2)

Agregando memoria a `PascalRápido`

- Con una celda, podemos lograr que `PascalRápido` cuente cuántas veces es invocada.
- La memoria (estado) es global en este caso.
- La memoria local a una función se denomina estado encapsulado.

Pascal con memoria

```
declare
C={NewCell 0}
fun {PascalRápido N}
   C:=@C+1
   {PascalGenérico Sum N}
end
```

Objetos

- Una función con memoria interna se llama objeto.
- Definamos un objeto contador, el cual tiene una celda que mantiene la cuenta actual.
- El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz.
- `Incr` agrega 1 a la cuenta y devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta.
- Encapsulación: la celda queda completamente invisible desde el exterior.
- Separar la interfaz de la implementación es la esencia de la abstracción de datos.

Objeto contador

```
declare
local C in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1 @C
   end
   fun {Leer}
      @C
   end
end
{Browse {Incr}}
{Browse {Incr}}
```

Objetos

- Una función con memoria interna se llama objeto.
- Definamos un objeto contador, el cual tiene una celda que mantiene la cuenta actual.
- El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz.
- `Incr` agrega 1 a la cuenta y devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta.
- Encapsulación: la celda queda completamente invisible desde el exterior.
- Separar la interfaz de la implementación es la esencia de la abstracción de datos.

Objeto contador

```
declare
local C in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1   @C
   end
   fun {Leer}
      @C
   end
end
{Browse {Incr}}
{Browse {Incr}}
```

Objetos

- Una función con memoria interna se llama objeto.
- Definamos un objeto contador, el cual tiene una celda que mantiene la cuenta actual.
- El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz.
- `Incr` agrega 1 a la cuenta y devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta.
- Encapsulación: la celda queda completamente invisible desde el exterior.
- Separar la interfaz de la implementación es la esencia de la abstracción de datos.

Objeto contador

```
declare
local C in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1   @C
   end
   fun {Leer}
      @C
   end
end
{Browse {Incr}}
{Browse {Incr}}
```

Objetos

- Una función con memoria interna se llama objeto.
- Definamos un objeto contador, el cual tiene una celda que mantiene la cuenta actual.
- El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz.
- `Incr` agrega 1 a la cuenta y devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta.
- Encapsulación: la celda queda completamente invisible desde el exterior.
- Separar la interfaz de la implementación es la esencia de la abstracción de datos.

Objeto contador

```
declare
local C in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1   @C
   end
   fun {Leer}
      @C
   end
end
{Browse {Incr}}
{Browse {Incr}}
```

Objetos

- Una función con memoria interna se llama objeto.
- Definamos un objeto contador, el cual tiene una celda que mantiene la cuenta actual.
- El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz.
- `Incr` agrega 1 a la cuenta y devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta.
- Encapsulación: la celda queda completamente invisible desde el exterior.
- Separar la interfaz de la implementación es la esencia de la abstracción de datos.

Objeto contador

```
declare
local C in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1  @C
   end
   fun {Leer}
      @C
   end
end
{Browse {Incr}}
{Browse {Incr}}
```

Objetos

- Una función con memoria interna se llama objeto.
- Definamos un objeto contador, el cual tiene una celda que mantiene la cuenta actual.
- El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz.
- `Incr` agrega 1 a la cuenta y devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta.
- Encapsulación: la celda queda completamente invisible desde el exterior.
- Separar la interfaz de la implementación es la esencia de la abstracción de datos.

Objeto contador

```
declare
local C in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1  @C
   end
   fun {Leer}
      @C
   end
end
{Browse {Incr}}
{Browse {Incr}}
```

Objetos

- Una función con memoria interna se llama objeto.
- Definamos un objeto contador, el cual tiene una celda que mantiene la cuenta actual.
- El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz.
- `Incr` agrega 1 a la cuenta y devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta.
- Encapsulación: la celda queda completamente invisible desde el exterior.
- Separar la interfaz de la implementación es la esencia de la abstracción de datos.

Objeto contador

```
declare
local C in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1  @C
   end
   fun {Leer}
      @C
   end
end
{Browse {Incr}}
{Browse {Incr}}
```

Clases (1)

Fábricas de objetos

- ¿Qué pasa si necesitamos más de un contador?
- Crear una “fábrica” que pueda fabricar tantos contadores como necesitemos.
- Tal fábrica se llama una clase.
- Se implementó con prog. de alto orden.

```
declare
fun {ContadorNuevo}
   C Incr Leer in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1
      @C
   end
   fun {Leer}
      @C
   end
   contador(incr:Incr
            leer:Leer)
end
```

Clases (1)

Fábricas de objetos

- ¿Qué pasa si necesitamos más de un contador?
- Crear una “fábrica” que pueda fabricar tantos contadores como necesitemos.
- Tal fábrica se llama una clase.
- Se implementó con prog. de alto orden.

```
declare
fun {ContadorNuevo}
   C Incr Leer in
      C={NewCell 0}
      fun {Incr}
         C:=@C+1   @C
      end
      fun {Leer}
         @C
      end
      contador(incr:Incr
               leer:Leer)
end
```

Clases (1)

Fábricas de objetos

- ¿Qué pasa si necesitamos más de un contador?
- Crear una “fábrica” que pueda fabricar tantos contadores como necesitemos.
- Tal fábrica se llama una clase.
- Se implementó con prog. de alto orden.

```
declare
fun {ContadorNuevo}
   C Incr Leer in
      C={NewCell 0}
      fun {Incr}
         C:=@C+1   @C
      end
      fun {Leer}
         @C
      end
      contador(incr:Incr
               leer:Leer)
end
```

Clases (1)

Fábricas de objetos

- ¿Qué pasa si necesitamos más de un contador?
- Crear una “fábrica” que pueda fabricar tantos contadores como necesitemos.
- Tal fábrica se llama una clase.
- Se implementó con prog. de alto orden.

```
declare
fun {ContadorNuevo}
C Incr Leer in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1   @C
   end
   fun {Leer}
      @C
   end
   contador(incr:Incr
            leer:Leer)
end
```

Clases (1)

Fábricas de objetos

- ¿Qué pasa si necesitamos más de un contador?
- Crear una “fábrica” que pueda fabricar tantos contadores como necesitemos.
- Tal fábrica se llama una clase.
- Se implementó con prog. de alto orden.

```
declare
fun {ContadorNuevo}
C Incr Leer in
   C={NewCell 0}
   fun {Incr}
      C:=@C+1   @C
   end
   fun {Leer}
      @C
   end
   contador(incr:Incr
            leer:Leer)
end
```

Clases (2)

Usando la fábrica

Creemos dos contadores:

```
declare
Ctr1={ContadorNuevo}
Ctr2={ContadorNuevo}
```

Hagamos crecer el primer contador y mostremos su resultado:

```
{Browse {Ctr1.incr}}
```

De la prog. basada en objetos a la POO

- Las operaciones definidas dentro de las clases se llaman métodos.
- La clase puede ser usada para crear tantos objetos contador como necesitemos.
- Los objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado.
- La programación con clases y objetos se llama programación basada en objetos (PBO).
- POO = PBO + herencia
- Herencia: definir una clase nueva en términos de clases ya existentes.
- La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial.

De la prog. basada en objetos a la POO

- Las operaciones definidas dentro de las clases se llaman métodos.
- La clase puede ser usada para crear tantos objetos contador como necesitemos.
- Los objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado.
- La programación con clases y objetos se llama programación basada en objetos (PBO).
- POO = PBO + herencia
- Herencia: definir una clase nueva en términos de clases ya existentes.
- La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial.

De la prog. basada en objetos a la POO

- Las operaciones definidas dentro de las clases se llaman métodos.
- La clase puede ser usada para crear tantos objetos contador como necesitemos.
- Los objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado.
- La programación con clases y objetos se llama programación basada en objetos (PBO).
- POO = PBO + herencia
- Herencia: definir una clase nueva en términos de clases ya existentes.
- La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial.

De la prog. basada en objetos a la POO

- Las operaciones definidas dentro de las clases se llaman métodos.
- La clase puede ser usada para crear tantos objetos contador como necesitemos.
- Los objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado.
- La programación con clases y objetos se llama programación basada en objetos (PBO).
- POO = PBO + herencia
- Herencia: definir una clase nueva en términos de clases ya existentes.
- La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial.

De la prog. basada en objetos a la POO

- Las operaciones definidas dentro de las clases se llaman métodos.
- La clase puede ser usada para crear tantos objetos contador como necesitemos.
- Los objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado.
- La programación con clases y objetos se llama programación basada en objetos (PBO).
- POO = PBO + herencia
- Herencia: definir una clase nueva en términos de clases ya existentes.
- La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial.

De la prog. basada en objetos a la POO

- Las operaciones definidas dentro de las clases se llaman métodos.
- La clase puede ser usada para crear tantos objetos contador como necesitemos.
- Los objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado.
- La programación con clases y objetos se llama programación basada en objetos (PBO).
- POO = PBO + herencia
- Herencia: definir una clase nueva en términos de clases ya existentes.
- La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial.

De la prog. basada en objetos a la POO

- Las operaciones definidas dentro de las clases se llaman métodos.
- La clase puede ser usada para crear tantos objetos contador como necesitemos.
- Los objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado.
- La programación con clases y objetos se llama programación basada en objetos (PBO).
- POO = PBO + herencia
- Herencia: definir una clase nueva en términos de clases ya existentes.
- La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial.

Universidad del Valle

Plan

No-Determinismo (1)

- ¿Qué pasa cuando un programa tiene concurrencia y estado al mismo tiempo?
- El mismo programa puede tener resultados diferentes entre una y otra ejecución.
- Esta variabilidad se llama no-determinismo.
- Problema cuando el no-determinismo es observable.

No-Determinismo (1)

- ¿Qué pasa cuando un programa tiene concurrencia y estado al mismo tiempo?
- El mismo programa puede tener resultados diferentes entre una y otra ejecución.
- Esta variabilidad se llama no-determinismo.
- Problema cuando el no-determinismo es observable.

No-Determinismo (1)

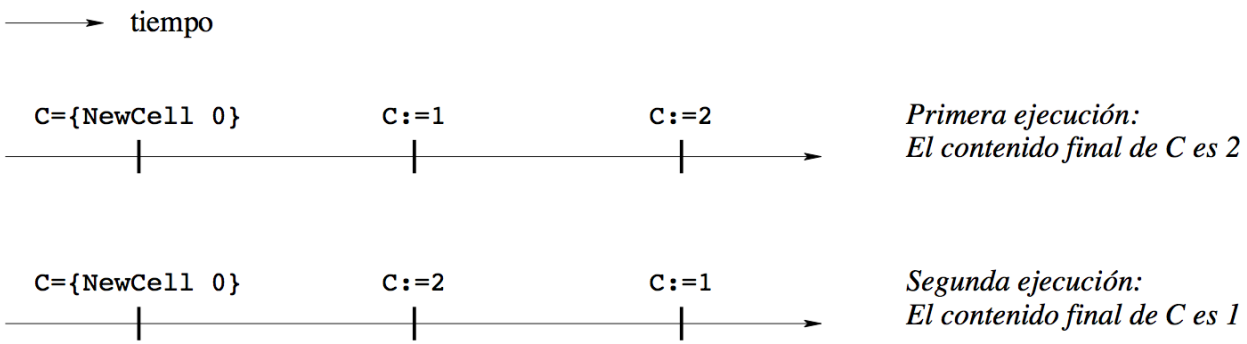
- ¿Qué pasa cuando un programa tiene concurrencia y estado al mismo tiempo?
- El mismo programa puede tener resultados diferentes entre una y otra ejecución.
- Esta variabilidad se llama no-determinismo.
- Problema cuando el no-determinismo es observable.

No-Determinismo (1)

- ¿Qué pasa cuando un programa tiene concurrencia y estado al mismo tiempo?
- El mismo programa puede tener resultados diferentes entre una y otra ejecución.
- Esta variabilidad se llama no-determinismo.
- Problema cuando el no-determinismo es observable.

No-Determinismo (2)

```
declare
C={NewCell 0}
thread
   C:=1
end
thread
   C:=2
end
```

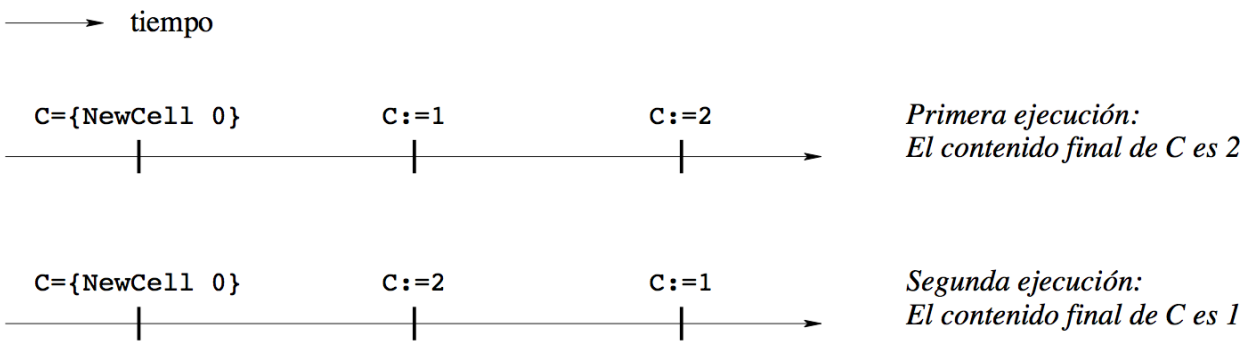


¿Cuál es el contenido de C después

que se ejecuta este programa?

No-Determinismo (2)

```
declare
C={NewCell 0}
thread
   C:=1
end
thread
   C:=2
end
```

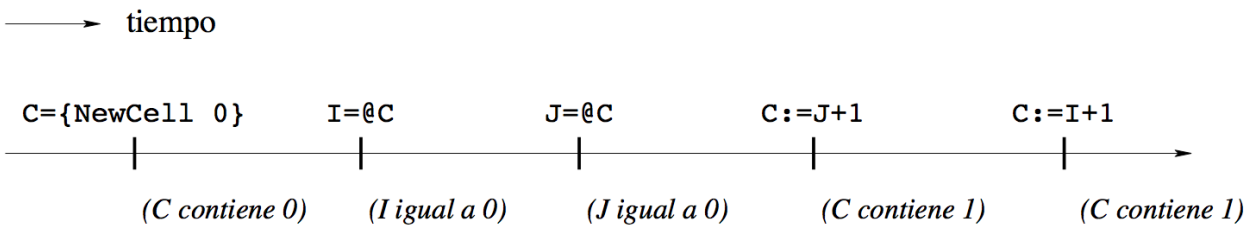


¿Cuál es el contenido de C después

que se ejecuta este programa?

No-Determinismo (3)

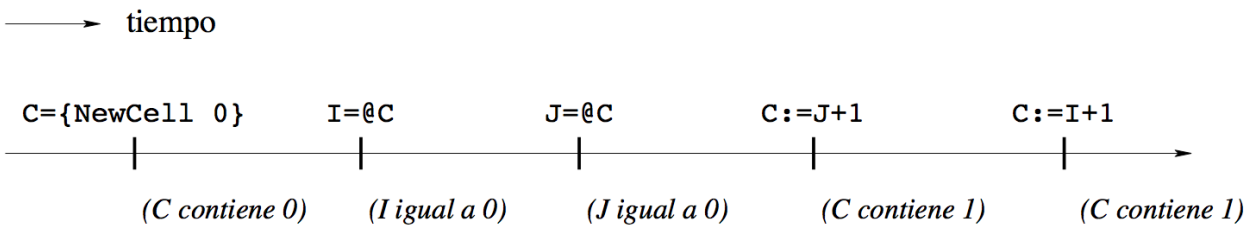
```
declare
C={NewCell 0}
thread I in
   I=@C C:=I+1
end
thread J in
   J=@C C:=J+1
end
```



¿Cuál es el contenido de C después que se ejecuta este programa?

No-Determinismo (3)

```
declare
C={NewCell 0}
thread I in
   I=@C C:=I+1
end
thread J in
   J=@C C:=J+1
end
```



¿Cuál es el contenido de C después que se ejecuta este programa?

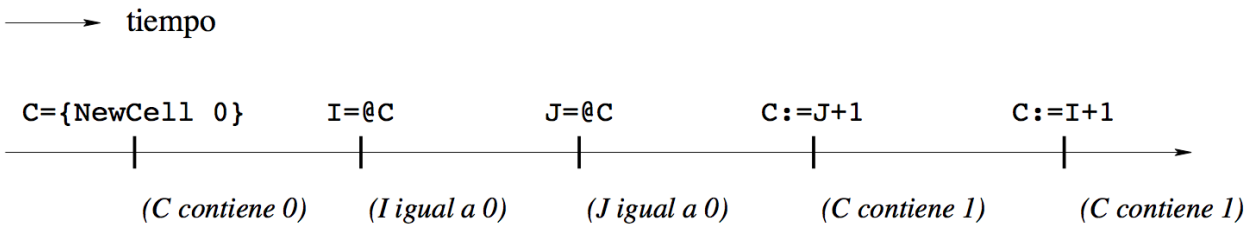
Intercalación

- El resultado puede ser 1 debido a una posible intercalación.
- Programar con concurrencia y estado juntos es principalmente un asunto de dominio de las intercalaciones.
- ¡De ser posible, no las use al mismo tiempo!

No-Determinismo (3)

```
declare
C={NewCell 0}
thread I in
   I=@C C:=I+1
end
thread J in
   J=@C C:=J+1
end
```

¿Cuál es el contenido de C después que se ejecuta este programa?



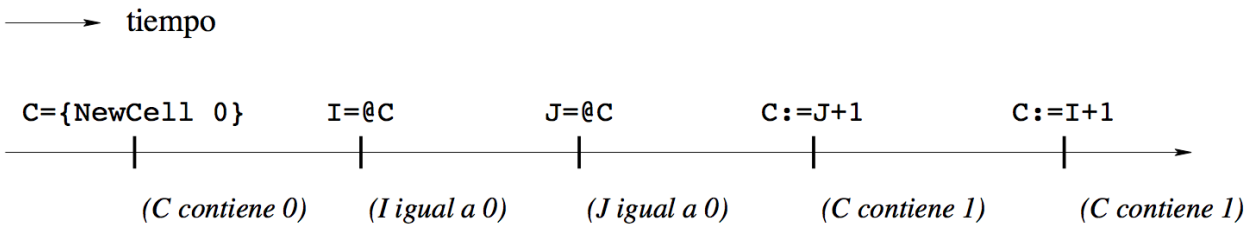
Intercalación

- El resultado puede ser 1 debido a una posible intercalación.
- Programar con concurrencia y estado juntos es principalmente un asunto de dominio de las intercalaciones.
- ¡de ser posible, no las use al mismo tiempo!

No-Determinismo (3)

```
declare
C={NewCell 0}
thread I in
   I=@C C:=I+1
end
thread J in
   J=@C C:=J+1
end
```

¿Cuál es el contenido de C después que se ejecuta este programa?



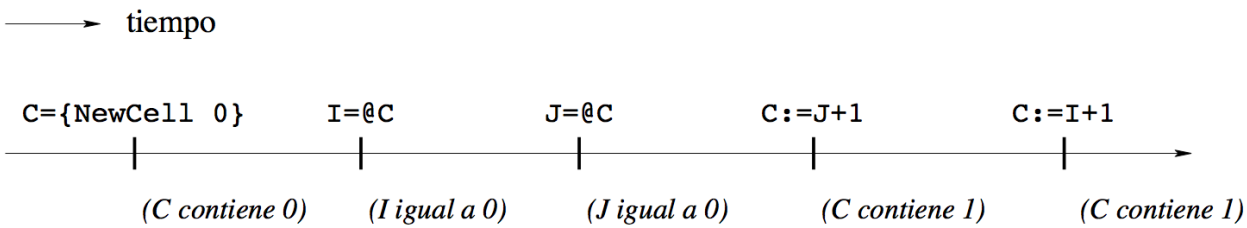
Intercalación

- El resultado puede ser 1 debido a una posible intercalación.
- Programar con concurrencia y estado juntos es principalmente un asunto de dominio de las intercalaciones.
- ¡de ser posible, no las use al mismo tiempo!

No-Determinismo (3)

```
declare
C={NewCell 0}
thread I in
   I=@C C:=I+1
end
thread J in
   J=@C C:=J+1
end
```

¿Cuál es el contenido de C después que se ejecuta este programa?



Intercalación

- El resultado puede ser 1 debido a una posible intercalación.
- Programar con concurrencia y estado juntos es principalmente un asunto de dominio de las intercalaciones.
- ¡de ser posible, no las use al mismo tiempo!

Para donde vamos

Estudio y práctica de diversos modelos de programación

- Declarativo
- Concurrente Declarativo
- Con estado explícito
- Concurrente por paso de mensajes
- Orientado a Objetos
- Relacional