Programación Funcional

Clases teóricas

por Pablo E. “Fidel” Martínez López

14. Lambda cálculo

"*El lujo es vulgaridad dijo, y me conquistó.*"

Un poco de amor francés
La mosca y la sopa, 1991

Patricio Rey y sus
Redonditos de Ricota

Motivación

Motivación

- ¿Cuál sería la cantidad mínima de conceptos necesaria para definir un lenguaje de programación (funcional)?

Motivación

- ¿Cuál sería la cantidad mínima de conceptos necesaria para definir un lenguaje de programación (funcional)?
  - Los lenguajes funcionales muestran que no es necesario el concepto de memoria
  - ¿Son necesarios los números? ¿Y los booleanos?
  - ¿Son necesarias las estructuras de datos (algebraicas)?
  - ¿Qué debe incluir un lenguaje minimalista?

Motivación

- ¿Cuál sería la cantidad mínima de conceptos necesaria para definir un lenguaje de programación (funcional)?
  - La respuesta es sorprendentemente simple

Motivación

- ¿Cuál sería la cantidad mínima de conceptos necesaria para definir un lenguaje de programación (funcional)?
  - La respuesta es sorprendentemente simple:
    Funciones (transformación de información)

Motivación

- ¿Cuál sería la cantidad mínima de conceptos necesaria para definir un lenguaje de programación (funcional)?
  - La respuesta es sorprendentemente simple: ***Funciones*** (transformación de información)
- ¿Se puede programar con un lenguaje que solamente tenga funciones?
  - ¿Qué información transformarían estas funciones?

Motivación

- ¿Cuál sería la cantidad mínima de conceptos necesaria para definir un lenguaje de programación (funcional)?
  - La respuesta es sorprendentemente simple:
    Funciones (transformación de información)
- ¿Se puede programar con un lenguaje que solamente tenga funciones?
  - ¿Qué información transformarían estas funciones?
    - ¡Otras funciones!

Motivación

- ¿Qué aspecto tendría este lenguaje?
  - ¿Qué símbolos usar? ***Sintaxis***
  - ¿Qué significados darles? ***Semántica***

Motivación

- ¿Qué aspecto tendría este lenguaje?
  - ¿Qué símbolos usar? ***Sintaxis***
  - ¿Qué significados darles? ***Semántica***
- En esta clase (por tiempos)
  - Breve desarrollo de sintaxis
  - Semántica manejada en forma intuitiva
  - Muestras de cómo expresar conceptos usuales de programación con este lenguaje

Lambda cálculo

Lambda cálculo

- ¿Con qué herramientas se expresan las funciones?
  - *Descripción* de funciones no conocidas
  - *Definición* de funciones
  - *Uso* de funciones

Lambda cálculo

- ¿Con qué herramientas se expresan las funciones?
  - *Descripción* de funciones no conocidas
  - *Definición* de funciones
  - *Uso* de funciones
- ¿Qué herramientas se usan para eso?
  - Variables
  - Funciones anónimas
  - Aplicación

Lambda cálculo

- ¿Qué sintaxis usar para estos elementos?
  - Variables
    - Cualquier conjunto (infinito) de **identificadores**
      - El requerimiento de ser infinito es por razones técnicas
  - Funciones anónimas
    - **Notación lambda**
  - Aplicación
    - **Yuxtaposición**

Lambda cálculo, sintaxis

- ¿Qué sintaxis usar para estos elementos?
  - Variables
    - Ej: x
      - En general: x, y, z, x_0, x_1, ...
  - Funciones anónimas
    - Ej: $(\lambda x.x)$
  - Aplicación
    - Ej: $((\lambda x.x)\ (\lambda x.x))$

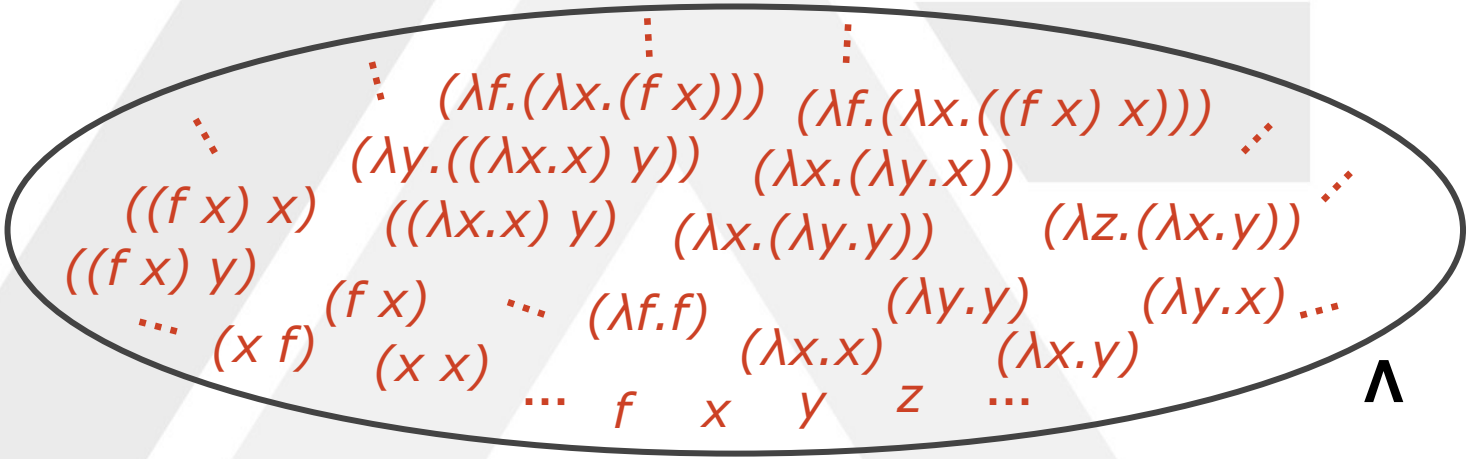
Lambda cálculo, sintaxis

- Definición del lenguaje como conjunto de expresiones
  - Se define el conjunto **Λ** por inducción estructural
    - Si **x** es una variable, entonces **x** es un elemento de **Λ**
      - Posible variables: x, y, z, x_0, x_1, ... (hay infinitas)
    - Si **x** es una variable, y **M** es un elemento de **Λ** entonces $(\lambda \mathbf{x}.\mathbf{M})$ es un elemento de **Λ**
    - Si **M** y **N** son dos elmentos de **Λ** entonces $(\mathbf{M}\ \mathbf{N})$ es un elemento de **Λ**
  - El conjunto **Λ** se denomina ***conjunto de expresiones lambda***

Lambda cálculo, sintaxis

- Conjunto de expresiones **Λ** por inducción estructural
  - Si **x** ∈ **V**, entonces **x** ∈ **Λ**
  - Si **x** ∈ **V**, y **M** ∈ **Λ** entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si **M** ∈ **Λ** y **N** ∈ **Λ** entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$



Lambda cálculo, sintaxis

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$

- Definición del lenguaje como conjunto de expresiones
  - Los programas del conjunto $\mathbf{\Lambda}$ son poco usuales
    - De interés
      $(\lambda w.w)$
      $(\lambda f.(\lambda x.(f\ x)))$
      $(\lambda x.(\lambda y.x))$
      $(\lambda f.(\lambda z.((f\ z)\ z)))$
      $((\lambda y.y)\ (\lambda y.y))$
    - Con menos interés
      $(f\ x)$
      $(\lambda x.y)$
      $(\lambda x.(\lambda y.z))$
      $((\lambda x.x)\ z)$
      $((\lambda x.y)\ (\lambda x.x))$
    - ¿Cuál es la diferencia entre ambos grupos?

Lambda cálculo

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$

- Definición del lenguaje como conjunto de expresiones
  - Los programas del conjunto $\mathbf{\Lambda}$ son poco usuales
    - De interés
      $(\lambda x.x)$
      $(\lambda f.(\lambda x.(f\ x)))$
      $(\lambda x.(\lambda y.x))$
      $(\lambda f.(\lambda x.((f\ x)\ x)))$
      $((\lambda x.x)\ (\lambda x.x))$
    - Con menos interés
      $(f\ x)$
      $(\lambda x.y)$
      $(\lambda x.(\lambda y.z))$
      $((\lambda x.x)\ z)$
      $((\lambda x.y)\ (\lambda x.x))$
    - ¿Cuál es la diferencia entre ambos grupos?
      - Las variables mencionadas aparecen en lambdas

Lambda cálculo, sintaxis

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$

- Definición del lenguaje como conjunto de expresiones
  - Es necesario definir algunas nociones sintácticas
    - Ligadura de variables y alcance
    - Variables libres y ligadas
    - Expresiones abiertas y cerradas

No se definirán en este curso

  - Solamente serán de interés las expresiones cerradas
    - **Intuición:** toda variable se usa bajo un lambda que la contiene

Lambda cálculo, semántica

V = { $x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots$ }

- Conjunto **Λ** por inducción
  - Si **x** ∈ **V**, entonces **x** ∈ **Λ**
  - Si **x** ∈ **V**, y **M** ∈ **Λ** entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si **M** ∈ **Λ** y **N** ∈ **Λ** entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$

- Definición del lenguaje como conjunto de expresiones
  - La aplicación de funciones se resuelve sintácticamente

$((\lambda x.x)\ (\lambda z.z))$ reduce a $(\lambda z.z)$

o sea, x, donde x se reemplazó por $(\lambda z.z)$

$((\lambda f.(\lambda x.(f\ x)))\ (\lambda z.z))$ reduce a $(\lambda x.((\lambda z.z)\ x))$

o sea, $(\lambda x.(f\ x))$, donde f se reemplazó por $(\lambda z.z)$

Lambda cálculo, semántica

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$

- Definición del lenguaje como conjunto de expresiones
  - La aplicación de funciones se resuelve sintácticamente

$((\lambda x.x)\ (\lambda z.z))$ reduce a $(\lambda z.z)$

o sea, x, donde x se reemplazó por $(\lambda z.z)$

$((\lambda f.(\lambda x.(f\ x)))\ (\lambda z.z))$ reduce a $(\lambda x.((\lambda z.z)\ x))$

o sea, $(\lambda x.(f\ x))$, donde f se reemplazó por $(\lambda z.z)$

  - Reglas de reducción
    - Se usará el símbolo $\rightarrow$ para la reducción

$((\lambda x.x)\ (\lambda z.z)) \rightarrow (\lambda z.z)$

$((\lambda f.(\lambda x.(f\ x)))\ (\lambda z.z)) \rightarrow (\lambda x.((\lambda z.z)\ x)) \rightarrow (\lambda x.x)$

No se definirán en este curso

Lambda cálculo, semántica

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$

- Definición del lenguaje como conjunto de expresiones
  - Las expresiones cerradas describen funciones
    - $(\lambda x.x)$ describe a la función que toma un elemento y lo devuelve sin modificaciones (la función identidad)
    - $(\lambda f.(\lambda x.(f\ x)))$ describe a la función que toma una función y un elemento, y aplica la función al elemento
    - $(\lambda x.(\lambda y.x))$ describe a la función que toma un elemento y devuelve una función constante que siempre lo devuelve
    - $(\lambda f.(\lambda x.((f\ x)\ x)))$ describe a la función que toma una función y un elemento, y aplica la función dos veces a ese elemento

Lambda cálculo

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Los nombres de las variables ligadas no son importantes
  - ¿Qué significan las siguientes expresiones?
    - $(\lambda x.x)$ describe a la función que toma un elemento y lo devuelve sin modificaciones
    - $(\lambda y.y)$ describe a la función que toma un elemento y lo devuelve sin modificaciones
    - $(\lambda z.z)$ describe a la función que toma un elemento y lo devuelve sin modificaciones
  - Todas describen a la misma función, la **función identidad**
  - Las variables ligadas se pueden *renombrar* si es necesario
  - Existen reglas sintácticas para definir esto
  - Se define una noción de equivalencia

No se definirán en este curso

Lambda cálculo, notación

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Podemos agregar nombres fuera del lenguaje
  - Mecanismo de **convención sintáctica**
    - *<nombre>* $=_{\mathbf{def}}$ *<expresión-lambda>*
    - Es similar al **`#define`** de C
    - Permite nombrar funciones con sentido
      - **??** $=_{\mathbf{def}} (\lambda x.x)$
      - **??** $=_{\mathbf{def}} (\lambda f.(\lambda x.(f\ x)))$
      - **??** $=_{\mathbf{def}} (\lambda x.(\lambda y.x))$
      - **??** $=_{\mathbf{def}} (\lambda f.(\lambda x.((f\ x)\ x)))$

Lambda cálculo, notación

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Podemos agregar nombres fuera del lenguaje
  - Mecanismo de **convención sintáctica**
    - *<nombre>* $=_{\mathtt{def}}$ *<expresión-lambda>*
    - Es similar al **`#define`** de C
    - Permite nombrar funciones con sentido
      - **id** $=_{\mathtt{def}} (\lambda x.x)$
      - **??** $=_{\mathtt{def}} (\lambda f.(\lambda x.(f\ x)))$
      - **??** $=_{\mathtt{def}} (\lambda x.(\lambda y.x))$
      - **??** $=_{\mathtt{def}} (\lambda f.(\lambda x.((f\ x)\ x)))$

También podría ser
id $=_{\mathtt{def}} (\lambda w.w)$

Lambda cálculo, notación

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Podemos agregar nombres fuera del lenguaje
  - Mecanismo de **convención sintáctica**
    - *<nombre>* $=_{\mathbf{def}}$ *<expresión-lambda>*
    - Es similar al **`#define`** de C
    - Permite nombrar funciones con sentido
      - **id** $=_{\mathbf{def}} (\lambda x.x)$
      - **apply** $=_{\mathbf{def}} (\lambda f.(\lambda x.(f\ x)))$
      - **??** $=_{\mathbf{def}} (\lambda x.(\lambda y.x))$
      - **??** $=_{\mathbf{def}} (\lambda f.(\lambda x.((f\ x)\ x)))$

También podría ser

apply $=_{\mathbf{def}} (\lambda x.(\lambda y.(x\ y)))$

Lambda cálculo, notación

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Podemos agregar nombres fuera del lenguaje
  - Mecanismo de **convención sintáctica**
    - *<nombre>* $=_{\mathbf{def}}$ *<expresión-lambda>*
    - Es similar al **`#define`** de C
    - Permite nombrar funciones con sentido
      - **id** $=_{\mathbf{def}} (\lambda x.x)$
      - **apply** $=_{\mathbf{def}} (\lambda f.(\lambda x.(f\ x)))$
      - **const** $=_{\mathbf{def}} (\lambda x.(\lambda y.x))$
      - **??** $=_{\mathbf{def}} (\lambda f.(\lambda x.((f\ x)\ x)))$

Lambda cálculo, notación

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Podemos agregar nombres fuera del lenguaje
  - Mecanismo de **convención sintáctica**
    - *<nombre>* $=_{\mathtt{def}}$ *<expresión-lambda>*
    - Es similar al **`#define`** de C
    - Permite nombrar funciones con sentido
      - **id** $=_{\mathtt{def}} (\lambda x.x)$
      - **apply** $=_{\mathtt{def}} (\lambda f.(\lambda x.(f\ x)))$
      - **const** $=_{\mathtt{def}} (\lambda x.(\lambda y.x))$
      - **appDup'** $=_{\mathtt{def}} (\lambda f.(\lambda x.((f\ x)\ x)))$

Lambda cálculo, notación

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Podemos agregar nombres fuera del lenguaje
  - Mecanismo de **convención sintáctica**
    - *<nombre>* $=_{\mathbf{def}}$ *<expresión-lambda>*
    - Es similar al **`#define`** de C
    - Permite nombrar funciones con sentido
      - **id** $=_{\mathbf{def}} (\lambda x.x)$
      - **apply** $=_{\mathbf{def}} (\lambda f.(\lambda x.(f\ x)))$
      - **const** $=_{\mathbf{def}} (\lambda x.(\lambda y.x))$
      - **appDup'** $=_{\mathbf{def}} (\lambda f.(\lambda x.((f\ x)\ x)))$

Las expresiones siguen siendo completamente rojas (las partes verdes deben reemplazarse por rojas)

Lambda cálculo, notación

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- ¿Cómo aliviar la notación? (Escribir sin tantos paréntesis)
  - Más **convenciones de notación** (similares a las de Haskell)
    - Los paréntesis externos pueden omitirse
      - Así $(\lambda x.(\lambda y.x))$ puede escribirse $\lambda x.\lambda y.x$
    - La aplicación asocia a izquierda
      - Así $f\ x\ y$ significa $((f\ x)\ y)$ y no $(f\ (x\ y))$
    - La aplicación tiene más precedencia que la abstracción
      - Así $\lambda f.f\ x$ significa $(\lambda f.(f\ x))$ y no $((\lambda f.f)\ x)$
    - Pueden juntarse varios lambda consecutivos
      - Así $(\lambda f.(\lambda x.(\lambda y.((f\ x)\ y))))$ puede escribirse $\lambda f\ x\ y.f\ x\ y$
    - Observar la posibilidad de leer “en francés”

Lambda cálculo

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

id	$=_{\mathbf{def}}\ \lambda x.x$
apply	$=_{\mathbf{def}}\ \lambda f\ x.f\ x$
const	$=_{\mathbf{def}}\ \lambda x\ y.x$
appDup'	$=_{\mathbf{def}}\ \lambda f\ x.f\ x\ x$

- Ejemplos (o cómo poner todo junto)
  - **id id** $=_{\mathbf{def}} (\lambda x.x)\ (\lambda z.z) \rightarrow (\lambda z.z) =_{\mathbf{def}}$ **id**
    - Por lo tanto **id id** reduce a **id**
    - Son expresiones equivalentes (significan lo mismo: la función identidad)
  - La noción de equivalencia se extiende a la reducción
  - Todas las expresiones equivalentes significan lo mismo

Lambda cálculo

$\mathbf{V} = \{ x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots \}$

- Conjunto Λ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \Lambda$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \Lambda$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \Lambda$
  - Si $\mathbf{M} \in \Lambda$ y $\mathbf{N} \in \Lambda$ entonces $(\mathbf{M}\ \mathbf{N}) \in \Lambda$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

- Ejemplos (o cómo poner todo junto)
  - **apply const id** $=_{\mathbf{def}}$ $(\lambda f.\lambda x.f\ x)$ **const id**
    $\rightarrow$ $(\lambda x.$**const** $x)$ **id**
    $\rightarrow$ **const id**
    $=_{\mathbf{def}}$ $(\lambda x.\lambda y.x)$ **id**
    $\rightarrow$ $\lambda y.$**id**

id	$=_{\mathbf{def}}$	$\lambda x.x$
apply	$=_{\mathbf{def}}$	$\lambda f\ x.f\ x$
const	$=_{\mathbf{def}}$	$\lambda x\ y.x$
appDup'	$=_{\mathbf{def}}$	$\lambda f\ x.f\ x\ x$

  - Por lo tanto **apply const id** reduce a $\lambda y.$**id** y son equivalentes
  - La expresión **apply const id** representa a la expresión lambda

    $(\lambda f.\lambda x.f\ x)\ (\lambda z.\lambda y.z)\ (\lambda w.w)$

    o sea $(((\lambda f.(\lambda x.(f\ x)))\ (\lambda z.(\lambda y.z)))\ (\lambda w.w))$

Lambda cálculo

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

$\mathbf{id} =_{\mathbf{def}} \lambda x.x$
$\mathbf{apply} =_{\mathbf{def}} \lambda f\ x.f\ x$
$\mathbf{const} =_{\mathbf{def}} \lambda x\ y.x$
$\mathbf{appDup'} =_{\mathbf{def}} \lambda f\ x.f\ x\ x$

- Ejemplos (o cómo poner todo junto)
  - $\mathbf{appDup'\ apply}\ z =_{\mathbf{def}} (\lambda f.\lambda x.f\ x\ x)\ \mathbf{apply}\ z$
    $\rightarrow (\lambda x.\mathbf{apply}\ x\ x)\ z$
    $\rightarrow \mathbf{apply}\ z\ z$
    $=_{\mathbf{def}} (\lambda f.\lambda x.f\ x)\ z\ z$
    $\rightarrow (\lambda x.z\ x)\ z$
    $\rightarrow z\ z$
    - Por lo tanto $\mathbf{appDup'\ apply}\ z$ reduce a $z\ z$ y significa lo mismo
    - ¿Cuál es la expresión lambda representada por $\mathbf{appDup'\ apply}\ z$?

Lambda cálculo

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

id	$=_{def}$	$\lambda x.x$
apply	$=_{def}$	$\lambda f\ x.f\ x$
const	$=_{def}$	$\lambda x\ y.x$
appDup	$=_{def}$	$\lambda f\ x.f\ x\ x$

- ¿Es suficiente con esto para programar?
  - Sí.
  - El lambda cálculo tiene el mismo poder computacional que cualquier lenguaje de programación de propósitos generales
    - Tesis de Church-Turing
  - Se usarán ejemplos para construir intuición sobre esto

Lambda cálculo

$\mathbf{V} = \{x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, f, g, \ldots\}$

- Conjunto $\mathbf{\Lambda}$ por inducción
  - Si $\mathbf{x} \in \mathbf{V}$, entonces $\mathbf{x} \in \mathbf{\Lambda}$
  - Si $\mathbf{x} \in \mathbf{V}$, y $\mathbf{M} \in \mathbf{\Lambda}$ entonces $(\lambda \mathbf{x}.\mathbf{M}) \in \mathbf{\Lambda}$
  - Si $\mathbf{M} \in \mathbf{\Lambda}$ y $\mathbf{N} \in \mathbf{\Lambda}$ entonces $(\mathbf{M}\ \mathbf{N}) \in \mathbf{\Lambda}$
- Ejecución por reducción: $\rightarrow$
- Significado: expresiones cerradas = funciones

id	$=_{def}$	$\lambda x.x$
apply	$=_{def}$	$\lambda f\ x.f\ x$
const	$=_{def}$	$\lambda x\ y.x$
appDup	$=_{def}$	$\lambda f\ x.f\ x\ x$

- ¿Por qué es interesante tener tan poco?
  - Permite definiciones simples
    - Por inducción estructural
  - Facilita el estudio de aspectos computacionales
    - Este cálculo mantiene las propiedades interesantes que aparecen en otros lenguajes de programación
  - Facilita la demostración de propiedades
    - ¿Se imaginan una demostración en un lenguaje con 30 formas diferentes de expresiones?

Breve historia del lambda cálculo y la programación (funcional)

Lambda cálculo, historia

- Breve historia de los lenguajes funcionales
  - ~1920 –

Haskell B. Curry

Haskell Brooks Curry
(12 de septiembre 1900 – 1 de septiembre 1982) fue un lógico y matemático estadounidense graduado de la Universidad de Harvard que alcanzó la fama por su trabajo en ***lógica combinatoria***. A pesar de que su trabajo se basó principalmente en un único artículo de Moses Schönfinkel, fue Curry el que hizo el desarrollo más importante.
También se lo conoce por la paradoja de Curry y por la *Correspondencia de Curry-Howard* (un vínculo profundo entre la lógica y la computación). Hay 3 lenguajes de programación nombrados en su honor, **Haskell**, **Brook** and **Curry**, así como el concepto de *currificación*, una técnica para aplicar funciones de orden superior.

Lambda cálculo, historia


Haskell
B. Curry

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)

Alonzo Church

Alonzo Church
(14 de junio 1903 – 11 de agosto 1995) es un matemático y lógico estadounidense, que trabajó en la Universidad de Princeton.
Es conocido por numerosos desarrollos teóricos en lógica matemática y teoría de la computación. En esta última es conocido por su desarrollo del lambda cálculo, un formalismo para computar mediante funciones; también es conocido por la llamada *Tesis de Church-Turing*, que conjetura sobre la naturaleza de las funciones efectivamente computables mediante algún mecanismo automático, por una forma de representar datos en el lambda cálculo llamada la codificación de Church y por un resultado sobre confluencia en el lambda cálculo, llamado el teorema de Church-Rosser, entre muchos otros desarrollos importantes.

Lambda cálculo, historia


Haskell
B. Curry


Alonzo
Church

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)

Lambda cálculo, historia


Haskell
B. Curry


Alonzo
Church

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)

~1930 – Kurt Gödel (incompletitud)

Alan M. Turing

Alan Matheson Turing
(23 de junio 1912 – 7 de junio 1954) es un matemático, lógico, criptoanalista, biólogo, filósofo y científico de la computación británico, de gran influencia en el desarrollo de la ciencia de la computación teórica. Desarrolló su tesis doctoral en computación bajo la dirección de Alonzo Church en la Universidad de Princeton.
Desarrolló las nociones de algoritmo y computación a través de un formalismo que él llamo “máquinas de computar”, y que Church rebautizó como “máquinas de Turing”, sentando las bases de una de las formas más difundidas para estudiar computabilidad. Mostró que sus máquinas eran un modelo de una computadora de propósitos generales, y demostró sus limitaciones teóricas (a través del llamado *Halting Problem*).

Lambda cálculo, historia


Haskell
B. Curry


Alonzo
Church

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)

~1930 – Kurt Gödel (incompletitud)

Lambda cálculo, historia


Haskell
B. Curry

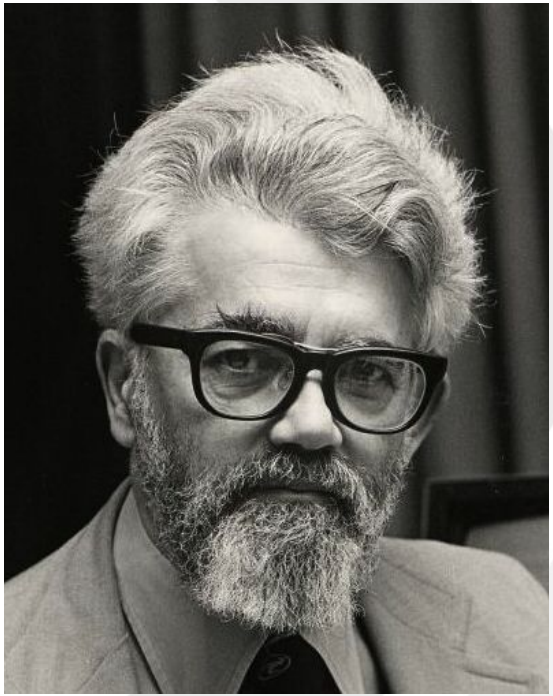

Alonzo
Church

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

John McCarthy



John McCarthy
(4 de septiembre 1927 – 24 de octubre 2011) es un científico de la computación estadounidense que desarrolló su carrera en la Universidad de Stanford. Tomó clases con John von Neumann en Caltech, quién lo inspiró en su carrera.
Desarrolló el lenguaje LISP en 1958, inventó el concepto de *garbage collection* en 1959, participó en la creación de ALGOL y también fue uno de los fundadores de la disciplina de Inteligencia Artificial (junto con Alan Turing, Marvin Minsky y otros), de la que es responsable por acuñar el término en 1955. Fue quién propuso el uso de recursión y de if-then-else en lenguajes de programación, e introdujo el uso de funciones de orden superior en programación para LISP.

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

Peter J. Landin

Peter John Landin
(5 de junio 1930 – 3 de junio 2009) es un científico de la computación británico graduado en la Universidad de Cambdrige. Entre 1960 y 1964 fue ayudante de Christopher Strachey y trabajó con Peter Naur y Edsger W. Dijkstra; en este período publicó la mayoría de su trabajo. Fue el primero en proponer el uso del lambda cálculo como modelo de lenguajes de programación, lo cual fue esencial tanto en el desarrollo de la programación funcional como de la semántica denotacional. Inventó la primera máquina abstracta para programación funcional, la *SECD machine*, y acuñó los términos *"syntactic sugar" y "off-side rule"* (para "indentación"). Promovió la enseñanza de algoritmos recursivos y de lenguajes sensibles a la indentación.

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy


Peter Landin

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (lambda cálculo para semántica)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy


Peter Landin

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (lambda cálculo para semántica)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

~1960 – Primeros conceptos de POO
Lenguaje C

Robin Milner

Arthur John Robin Gorell Milner
(13 de enero 1934 – 20 de marzo 2010) es un científico de la computación británico, fundador del Laboratorio de Fundamentos de Ciencias de la Computación (LFCS) de la Universidad de Edimburgo.
Desarrolló la *Lógica para Funciones Computables* (***LCF***) en 1972, una de las primeras herramientas para demostración automatizada de teoremas, el lenguaje ***ML*** (Meta-Language) en 1973, primer lenguaje con inferencia de tipos polimórfica (el sistema de tipos Hindley-Milner), pensado para implementar la LCF, y el *Cálculo de Sistemas Comunicantes* (***CCS***) en 1980 y su sucesor, el **π-cálculo** en 1992, para analizar sistemas concurrentes.

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy


Peter Landin


Robin Milner

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (lambda cálculo para semántica)
  - ~1970 – Robin Milner (ML, polimorifsmo)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

~1960 – Primeros conceptos de POO
Lenguaje C

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy


Peter Landin


Robin Milner

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (lambda cálculo para semántica)
  - ~1970 – Robin Milner (ML, polimorfismo)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

~1960 – Primeros conceptos de POO
Lenguaje C

~1980 – Smalltalk

R. John M. Hughes

R. John M. Hughes
(15 de julio 1958 – ...) es un científico de la computación, profesor de la Universidad de Tecnología de Chalmers, Gotemburgo, Suecia, y cofundador y CEO de QuviQ AB, empresa que ofrece una versión comercial de QuickCheck, de la que es también uno de los autores.
En 1984 recibió su doctorado de la Universidad de Oxford con la tesis *"The design and implementation of Programming Languages"*. Miembro del grupo de Programación Funcional en Chalmers, es uno de los miembros del comité que desarrolló el lenguaje Haskell. Dirigió numerosos doctorados de personalidades de PF, y de algún otro que no tanto.

Simon L. Peyton Jones

Simon L. Peyton Jones
(18 de enero 1958 – ...) es un científico de la computación británico, que investiga la implementación y aplicaciones de lenguajes funcionales, en particular lenguajes no estrictos. En 1980 se graduó en la Universidad de Cambridge (Trinity College) y enseñó en el University College de Londres y en la Universidad de Glasgow en Inglaterra. Desde 1998 trabaja para el grupo de investigación de Microsoft Research en Cambdrige, Inglaterra. Es uno de los miembros del comité que desarrolló el lenguaje Haskell, y uno de sus principales desarrolladores, siendo el diseñador principal del Glasgow Haskell Compiler (GHC) y autor de numerosísmos trabajos en el área.

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy


Peter Landin


Robin Milner


John Hughes & Simon Peyton Jones

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (lambda cálculo para semántica)
  - ~1970 – Robin Milner (ML, polimorifsmo)
  - ~1980 – John Hughes, Simon Peyton Jones (Haskell)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

~1960 – Primeros conceptos de POO
Lenguaje C

~1980 – Smalltalk

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy


Peter Landin


Robin Milner


John Hughes & Simon Peyton Jones

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (lambda cálculo para semántica)
  - ~1970 – Robin Milner (ML, polimorifsmo)
  - ~1980 – John Hughes, Simon Peyton Jones (Haskell)

~1930 – Kurt Gödel (incompletitud)

~1940 – John von Neumann (Arquitectura)
Primeras computadoras

~1960 – Primeros conceptos de POO
Lenguaje C

~1980 – Smalltalk

~1990 – Java

Martin Odersky

Martin Odersky
(5 de septiembre 1958 – ...) es un científico de la computación alemán, profesor de programación en la Escuela Politécnica Federal de Lausanne, en Suiza. Se recibió de doctor en el Instituto Federal Suizo de Tecnología, en Zurich, dirigido por Niklas Wirth, y posdoctorados en IBM y la Universidad de Yale.
Desarrolló el lenguaje de programación Scala, y también la versión Generic Java junto a Philip Wadler y otros (que incorporó la noción de *generics* a Java). También implementó el compilador GJ, que se transformó en la base del compilador de Java, `javac`.
Dicta dos MOOCs en Coursera sobre *Principios de Programación Funcional en Scala* y *Diseño de Programas Funcionales en Scala*.

Philip Wadler

Philip Lee Wadler
(8 de abril 1956 – ...) es un científico de la computación norteamericano, profesor de la Universidad de Edinburgo, en Inglaterra.
En 1984 recibió su doctorado de la Universidad de Carnegie Mellon y dedicó su carrera al estudio de la programación funcional. Trabajó en numerosas universidades y en Bell Labs, y actualmente es miembro de la empresa IOHK. Entre 1990 y 2004 fue editor del *Journal of Functional Programming*, una prestigiosa revista del área. Fue parte del grupo que desarrolló el lenguaje Haskell y estuvo involucrado en la incorporación de generics en el lenguaje Java. Es un excelente docente, muy histriónico y claro. En 2019 visitó la UNQ, donde dio la charla “Programming Language Foundations in Agda”.

Lambda cálculo, historia


Haskell B. Curry


Alonzo Church


John Mc Carthy


Peter Landin


Robin Milner


John Hughes & Simon Peyton Jones


Martin Odersky & Philip Wadler

- Breve historia de los lenguajes funcionales
  - ~1920 – Haskell B. Curry (lógica combinatoria, 1929)
  - ~1930 – Alonzo Church (lambda cálculo, 1933)
    Alan Turing (Máquina de computar, 1936)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (lambda cálculo para semántica)
  - ~1970 – Robin Milner (ML, polimorifsmo)
  - ~1980 – John Hughes, Simon Peyton Jones (Haskell)
  - ~2000 – Martin Odersky, Philip Wadler (Generics en Java)
    Martin Odersky (Scala)

~1930 – Kurt Gödel (incompletitud)

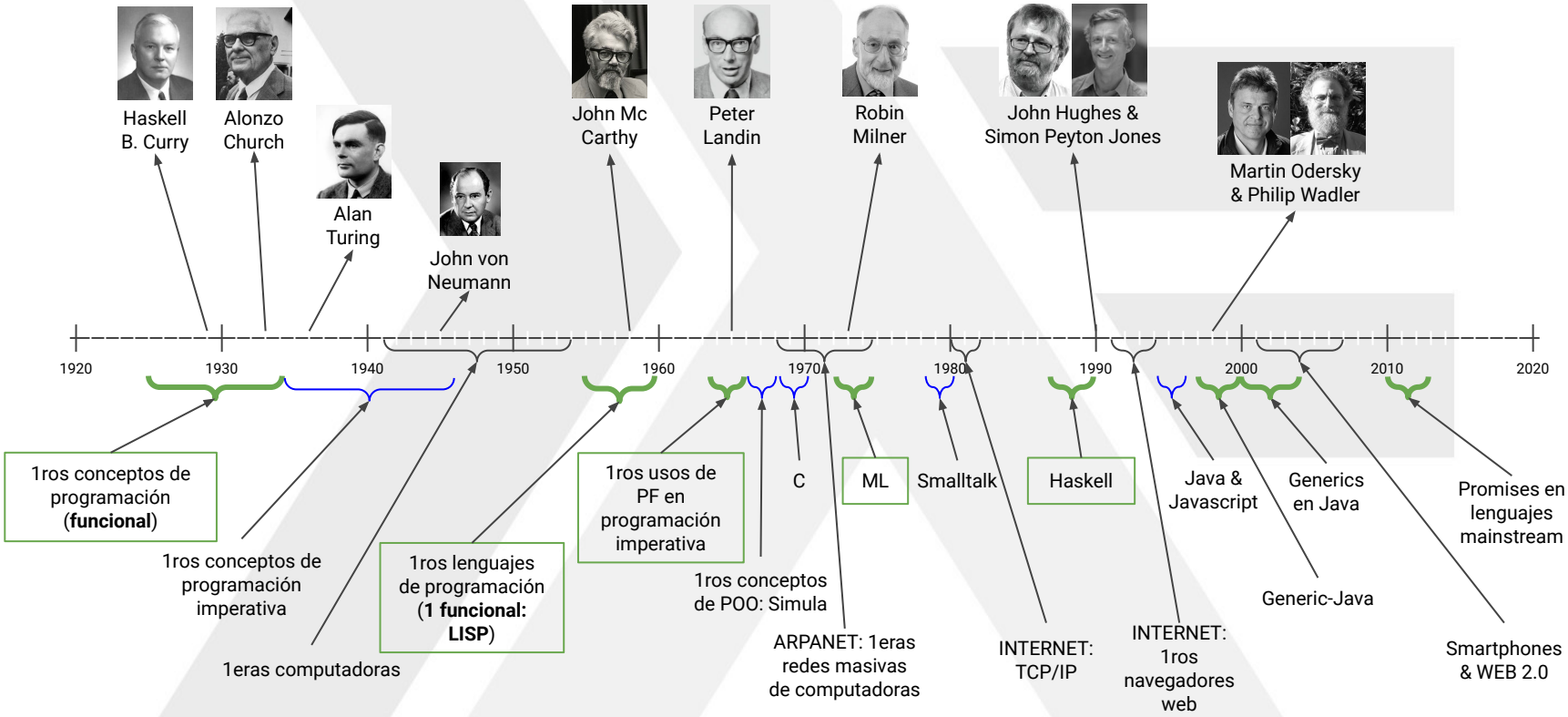
~1940 – John von Neumann (Arquitectura)
Primeras computadoras

~1960 – Primeros conceptos de POO
Lenguaje C

~1980 – Smalltalk

~1990 – Java

Casi 100 años de lenguajes funcionales



Lambda cálculo

- Usos del lambda cálculo (también llamado λ-cálculo)
  - Para compilación de lenguajes funcionales
  - Para dar semántica a lenguajes imperativos
    - e.g. Algol, LIS
  - Como formalismo para definir otras teorías
    - e.g. lógicas, sistemas de reescritura
      - $(\forall x.P(x))$ se representaría como $\forall(\lambda x.P)$
  - Como inspiración para otros cálculos
    - e.g. π-cálculo (concurrencia), σ-cálculo (objetos)

Representación de tipos en lambda cálculo

Lambda cálculo, programación

- ¿Es suficiente el λ-cálculo para programar?
  - Sí. Para mostrarlo, se representarán tipos de datos elementales con expresiones lambda
- ¿Qué significa representar un tipo en λ-cálculo?
  - Se establece qué propiedades deben cumplirse para ese tipo (especificación)
  - Se establece qué forma tienen:
    - las expresiones que representan elementos y operaciones del tipo, de tal forma que respeten la especificación

Lambda cálculo, programación

- Se representarán
  - Booleanos
  - Pares
  - Listas
  - Árboles (e.g. ExpA)
  - Números
  - Bottom
  - Recursión general

Lambda cálculo, programación

- Observaciones
  - El lenguaje base no tiene tipos, pero se supondrá que las operaciones representadas sí los tienen
  - No será tenido en cuenta el significado de expresiones que no respetan las reglas de formación
    - Ej: (**not 2**) será una expresión válida, pero no será tenida en cuenta a la hora de representar
  - El tratamiento de tipos es tema para otro curso

Booleanos

Lambda cálculo, booleanos

- **Booleanos**: especificación

$$\begin{aligned} \textbf{True} &=_{\texttt{def}} \ldots \\ \textbf{False} &=_{\texttt{def}} \ldots \\ \textbf{ifthenelse} &=_{\texttt{def}} (\lambda b.\lambda m.\lambda n.\ \ldots) \end{aligned}$$

tal que para todo par de expresiones **M** y **N**

$$\textbf{ifthenelse True } \mathbf{M\ N} \rightarrow \mathbf{M}$$
$$\textbf{ifthenelse False } \mathbf{M\ N} \rightarrow \mathbf{N}$$

Lambda cálculo, booleanos

True $=_{def}$ ...
False $=_{def}$ ...
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ ...)$

tal que para todo par de expresiones **M** y **N**
- **ifthenelse True** **M N** → **M**
- **ifthenelse False** **M N** → **N**

- ¿Cómo saber qué expresiones cumplen lo pedido?
  - Se usarán las propiedades pedidas para deducir
- ¿Cuántas formas hay de reemplazar las incógnitas?
  - ¡Hay infinitas combinaciones posibles!
  - Cualquier grupo de expresiones que cumplan, sirve
  - Debe elegirse una, y mantener esa
  - Observaciones:
    - Diferentes elecciones son incompatibles entre sí
    - Los booleanos sirven para elegir entre 2 alternativas
    - La construcción *if* es representable como función

Lambda cálculo, booleanos

True $=_{def}$ ...
False $=_{def}$ ...
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ ...)$

tal que para todo par de expresiones **M** y **N**
- **ifthenelse True** **M N** → **M**
- **ifthenelse False** **M N** → **N**

- **Booleanos:** deducir desde la especificación
  - Las propiedades pedidas, con paréntesis

    (ifthenelse True) **M N** → **M**
    (ifthenelse False) **M N** → **N**

    - Las expresiones en verde son funciones, ¿cuáles?

      ifthenelse True → $(\lambda x\ y.x)$
      ifthenelse False → $(\lambda x\ y.y)$

      - Toman 2 argumentos y devuelven uno de ellos
      - Es claro que cumplen la especificación original

Lambda cálculo, booleanos

True $=_{def}$ ...
False $=_{def}$ ...
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ ...)$

tal que para todo par de expresiones **M** y **N**
- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

- **Booleanos:** deducir desde la especificación, 2
  - Dos ecuaciones, tres incógnitas
  - Solución más simple: *decidir* que **ifthenelse** es la identidad

id True → $(\lambda x\ y.x)$
id False → $(\lambda x\ y.y)$

    - ¿Qué deben ser **True** y **False** en este caso?
      - ¡Los resultados!

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$

Lambda cálculo, booleanos

True $=_{def}$ ...
False $=_{def}$ ...
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ ...)$

tal que para todo par de expresiones **M** y **N**

- **ifthenelse True** **M N** → **M**
- **ifthenelse False** **M N** → **N**

- **Booleanos**: definición

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

- Y es fácil comprobar que se cumple que

ifthenelse True **M N** → **M**
ifthenelse False **M N** → **N**

Lambda cálculo, booleanos

True	$=_{def}$	$(\lambda x\ y.x)$
False	$=_{def}$	$(\lambda x\ y.y)$
ifthenelse	$=_{def}$	$(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

- **Booleanos**: comprobación de especificación

ifthenelse True M N

$=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$ **True M N**

→ $(\lambda m.\lambda n.$**True** $m\ n)$ **M N**

→ **True M N**

$=_{def}$ $(\lambda x\ y.x)$ **M N**

→ $(\lambda y.$**M**$)$ **N**

→ **M**

ifthenelse False M N

$=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$ **False M N**

→ $(\lambda m.\lambda n.$**False** $m\ n)$ **M N**

→ **False M N**

$=_{def}$ $(\lambda x\ y.y)$ **M N**

→ $(\lambda y.y)$ **N**

→ **N**

Lambda cálculo, booleanos

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** $\rightarrow$ **M**
- **ifthenelse False M N** $\rightarrow$ **N**

- **Booleanos**: observaciones e intuición
  - ¡Los valores de **True** y **False** son funciones!
    - ¿Podía ser de otra manera?
  - Intuición
    - El valor **True** es la capacidad de elegir por la primera alternativa
    - El valor **False** es la capacidad de elegir por la segunda alternativa
    - El operador **ifthenelse** funciona como despachador: simplemente delega el trabajo en los valores booleanos
      - La idea es similar a *double dispatch* en OOP

Lambda cálculo, booleanos

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True** **M N** → **M**
- **ifthenelse False** **M N** → **N**

- **Booleanos**: alternativa 1 - invertir los valores

Alternativa 1

True' $=_{def}$ $(\lambda x\ y.y)$
False' $=_{def}$ $(\lambda x\ y.x)$
ifthenelse' $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ \ldots)$

Observar la inversión

- ¿Qué debe ser **ifthenelse'** para cumplir la especificación?

ifthenelse' True' **M N** → **M**
ifthenelse' False' **M N** → **N**

Lambda cálculo, booleanos

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

- **Booleanos**: alternativa 1 - invertir los valores

Alternativa 1

True' $=_{def}$ $(\lambda x\ y.y)$
False' $=_{def}$ $(\lambda x\ y.x)$
ifthenelse' $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ b\ n\ m)$

- ¡Debe invertir el orden de aplicación!
  - La doble inversión obtiene el efecto buscado
  - Solamente debe elegirse UNA
  - Se preferirá la canónica

Lambda cálculo, booleanos

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**
- **ifthenelse True M N** $\rightarrow$ **M**
- **ifthenelse False M N** $\rightarrow$ **N**

- **Booleanos**: alternativa 2 - agregar parámetros

Alternativa 2

True'' $=_{def}$ $(\lambda x\ y\ z.x)$
False'' $=_{def}$ $(\lambda x\ y\ z.y)$
ifthenelse'' $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ \ldots)$

Observar el parámetro extra

- ¿Qué debe ser **ifthenelse''** para cumplir la especificación?

ifthenelse'' True'' **M N** $\rightarrow$ **M**
ifthenelse'' False'' **M N** $\rightarrow$ **N**

Lambda cálculo, booleanos

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

- **Booleanos**: alternativa 2 - agregar parámetros

Alternativa 2

True'' $=_{def}$ $(\lambda x\ y\ z.x)$
False'' $=_{def}$ $(\lambda x\ y\ z.y)$
ifthenelse'' $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ b\ m\ n\ \ldots)$

¡Hay infinitas opciones para completar esta expresión!

- Con esta forma ya hay infinitas opciones
  - El parámetro extra simplemente se ignora
  - Por eso puede ser cualquier expresión

Lambda cálculo, booleanos

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

- **Booleanos**: alternativa 2 - agregar parámetros

Alternativa 2

True'' $=_{def}$ $(\lambda x\ y\ z.x)$
False'' $=_{def}$ $(\lambda x\ y\ z.y)$
ifthenelse'' $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ b\ m\ n\ (\lambda x.x))$

Por ejemplo, la función identidad

- Con esta forma ya hay infinitas opciones
  - El parámetro extra simplemente se ignora
  - Por eso puede ser cualquier expresión

Lambda cálculo, booleanos

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** $\rightarrow$ **M**
- **ifthenelse False M N** $\rightarrow$ **N**

- **Booleanos**: alternativa 3 - agregar parámetros e invertir

Alternativa 3

True''' $=_{def}$ $(\lambda x\ y\ z.z)$
False''' $=_{def}$ $(\lambda x\ y\ z.x)$
ifthenelse''' $=_{def}$ $(\lambda b.\lambda m.\lambda n.\ b\ n\ (\lambda x.x)\ m)$

¡Atención a los nuevos cambios!

- También se puede
  - Agregar más parámetros
  - Cambiar el orden de selección de cada valor booleano
  - Así se obtienen infinitas variantes

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True** **M N** → **M**
- **ifthenelse False** **M N** → **N**

Lambda cálculo, booleanos

- **Booleanos**: otras operaciones
  - Se definen usando **ifthenelse**
  - Ejemplo: la conjunción

and $=_{def}$ $(\lambda b_1\ b_2.\ \ldots)$

tal que para todo booleano **B**

and True **B** → **B**
and False **B** → **False**

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

Lambda cálculo, booleanos

- **Booleanos**: otras operaciones
  - Se definen usando **ifthenelse**
  - Ejemplo: la conjunción

and $=_{def}$ $(\lambda b_1\ b_2.$ **ifthenelse** $b_1\ b_2$ **False**$)$

y se cumple que para todo booleano **B**

and True B → **B**
and False B → **False**

Lambda cálculo, booleanos

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**

- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

- **Booleanos**: otras operaciones
  - Se utilizará notación infija como convención

and $=_{def}$ $(\lambda b_1\ b_2.$ **if** b_1 **then** b_2 **else False**$)$

  - Se puede comprobar que se cumple la especificación

and True M N
$=_{def}$ $(\lambda b_1.\lambda b_2.$**if** b_1 **then** b_2 **else False**$)$ **True B**
→ **if True then B else False**
→ **B**

and False M N
$=_{def}$ $(\lambda b_1.\lambda b_2.$**if** b_1 **then** b_2 **else False**$)$ **False B**
→ **if False then B else False**
→ **False**

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**
- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

Lambda cálculo, booleanos

- **Booleanos**: otras operaciones
  - Con la representación canónica quedaría

    and $=_{def}$ $(\lambda b_1\ b_2.\ b_1\ b_2\ (\lambda x\ y.y))$
  - Con la alternativa 1 quedaría

    and' $=_{def}$ $(\lambda b_1\ b_2.\ b_1\ (\lambda x\ y.y)\ b_2)$
  - Todas cumplen la especificación
    - Es preferible con **ifthenelse**, pues funciona para todas
      - Es más abstracta

Canónica

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**
- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

Lambda cálculo, booleanos

- **Booleanos**: otras operaciones
  - Definir operaciones para **not** a **or**

not $=_{def}$ $(\lambda b.\ \ldots)$

tal que

not True → **False**
not False → **True**

or $=_{def}$ $(\lambda b_1\ b_2.\ \ldots)$

tal que para todo booleano **B**

or True B → **True**
or False B → **B**

  - **Recordar:** se definen usando **ifthenelse**

Lambda cálculo, booleanos

True $=_{def}$ $(\lambda x\ y.x)$
False $=_{def}$ $(\lambda x\ y.y)$
ifthenelse $=_{def}$ $(\lambda b.\lambda m.\lambda n.b\ m\ n)$

y es cierto que se cumple que para todas **M** y **N**
- **ifthenelse True M N** → **M**
- **ifthenelse False M N** → **N**

- **Booleanos**: cierre
  - Representación de datos como funciones
  - Los datos se representan como la función que establece cómo debe ser usado cada dato
  - El operador de acceso delega el trabajo en los datos
  - Los booleanos simplemente expresan la capacidad de elegir entre 2 alternativas que aún no se conocen
    - **True** es elegir la primera alternativa
    - **False** es elegir la segunda alternativa

Pares

Lambda cálculo, pares

- **Pares**: especificación

$$\begin{aligned} \mathbf{pair} &=_{\mathtt{def}} (\lambda x\ y.\ \ldots) \\ \mathbf{fst} &=_{\mathtt{def}} (\lambda p.\ \ldots) \\ \mathbf{snd} &=_{\mathtt{def}} (\lambda p.\ \ldots) \end{aligned}$$

tal que para todo par de expresiones **M** y **N**

$$\begin{aligned} \mathbf{fst}\ (\mathbf{pair}\ \mathbf{M}\ \mathbf{N}) &\rightarrow \mathbf{M} \\ \mathbf{snd}\ (\mathbf{pair}\ \mathbf{M}\ \mathbf{N}) &\rightarrow \mathbf{N} \end{aligned}$$

Lambda cálculo, pares

pair $=_{def}$ $(\lambda x\ y.\ \ldots)$
fst $=_{def}$ $(\lambda p.\ \ldots)$
snd $=_{def}$ $(\lambda p.\ \ldots)$

tal que para todo par de expresiones **M** y **N**

- **fst** $(\textbf{pair}\ M\ N) \rightarrow M$
- **snd** $(\textbf{pair}\ M\ N) \rightarrow N$

- Las ecuaciones son similares a las de los booleanos
  - Pero no son iguales… ¿Qué cambió?
  - El orden entre conocer y elegir
- ¿Cómo aprovechar esta similitud?
  - Usar booleanos, y usar parámetros para esperar la decisión
  - El constructor **pair** podría usar un **ifthenelse** para elegir entre los componentes del par, en base a un parámetro
  - Las funciones de acceso **fst** y **snd** proveerían el argumento necesario para satisfacer la especificación

Lambda cálculo, pares

pair $=_{def}$ $(\lambda x\ y.\ \ldots)$
fst $=_{def}$ $(\lambda p.\ \ldots)$
snd $=_{def}$ $(\lambda p.\ \ldots)$

tal que para todo par de expresiones **M** y **N**

- **fst** $(\textbf{pair}\ \textbf{M}\ \textbf{N}) \rightarrow \textbf{M}$
- **snd** $(\textbf{pair}\ \textbf{M}\ \textbf{N}) \rightarrow \textbf{N}$

- **Pares**: definición

pair $=_{def}$ $(\lambda x\ y.\lambda b.\ \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y)$
fst $=_{def}$ $(\lambda p.\ p\ \textbf{True})$
snd $=_{def}$ $(\lambda p.\ p\ \textbf{False})$

cumple que para todo par de expresiones **M** y **N**

$\textbf{fst}\ (\textbf{pair}\ \textbf{M}\ \textbf{N}) \rightarrow \textbf{M}$
$\textbf{snd}\ (\textbf{pair}\ \textbf{M}\ \textbf{N}) \rightarrow \textbf{N}$

- Comprobarlo

Lambda cálculo, pares

pair $=_{def}$ $(\lambda x\ y.\ \lambda b.$ **if** b **then** x **else** $y)$
fst $=_{def}$ $(\lambda p.\ p$ **True**$)$
snd $=_{def}$ $(\lambda p.\ p$ **False**$)$

cumple que para todo par de expresiones **M** y **N**

- **fst** (**pair M N**) → **M**
- **snd** (**pair M N**) → **N**

- Observar que el par (**pair M N**) es una función
  - ¿Debería sorprender?
- ¿Cómo se representa el par `(True,(&&))` de Haskell?

pair True and

$=_{def}$ $(\lambda x\ y.\lambda b.$ **if** b **then** x **else** $y)$ **True and**

→ $(\lambda b.$ **if** b **then True else and**$)$

$=_{def}$ $(\lambda b.\underbrace{(\lambda b'\ m\ n.\ b'\ m\ n)}_{\textbf{ifthenelse}}\ b\ \underbrace{(\lambda x\ y.x)}_{\textbf{True}}\ \underbrace{(\lambda b_1\ b_2.\ b_1\ b_2\ (\lambda x'\ y'.y'))}_{\textbf{and}})$

pair True and
= $(\lambda b.$ **if** b **then True else and**$)$

→ $(\lambda b.b\ (\lambda x\ y.x)\ (\lambda b_1\ b_2.\ b_1\ b_2\ (\lambda x'\ y'.y')))$

Código lambda para `(True,(&&))` *"Matrix-like"*

Lambda cálculo, pares

pair $=_{def}$ $(\lambda x\ y.\ \lambda b.$ **if** b **then** x **else** $y)$
fst $=_{def}$ $(\lambda p.\ p$ **True**$)$
snd $=_{def}$ $(\lambda p.\ p$ **False**$)$

cumple que para todo par de expresiones **M** y **N**

- **fst** $($**pair M N**$) \rightarrow$ **M**
- **snd** $($**pair M N**$) \rightarrow$ **N**

- ¿Y para tuplas de más elementos?
  - Se pueden usar pares de pares
  - Se puede repetir el proceso para más elementos

terna $=_{def}$ $(\lambda x\ y\ z.\ \ldots)$
fst3 $=_{def}$ $(\lambda t.\ \ldots)$
snd3 $=_{def}$ $(\lambda t.\ \ldots)$
thd3 $=_{def}$ $(\lambda t.\ \ldots)$

tal que para toda terna de expresiones **M**, **N** y **P**

fst3 $($**terna M N P**$) \rightarrow$ **M**
snd3 $($**terna M N P**$) \rightarrow$ **N**
thd3 $($**terna M N P**$) \rightarrow$ **P**

  - ¿Qué se podría usar para representar?

Lambda cálculo, pares

pair $=_{def}$ $(\lambda x\ y.\ \lambda b.$ **if** b **then** x **else** $y)$
fst $=_{def}$ $(\lambda p.\ p$ **True**$)$
snd $=_{def}$ $(\lambda p.\ p$ **False**$)$

cumple que para todo par de expresiones **M** y **N**

- **fst** $($**pair M N**$) \rightarrow$ **M**
- **snd** $($**pair M N**$) \rightarrow$ **N**

- ¿Y para tuplas de más elementos?
  - Se pueden usar pares de pares
  - Se puede repetir el proceso para más elementos

terna $=_{def}$ $(\lambda x\ y\ z.\ \ldots)$
fst3 $=_{def}$ $(\lambda t.\ \ldots)$
snd3 $=_{def}$ $(\lambda t.\ \ldots)$
thd3 $=_{def}$ $(\lambda t.\ \ldots)$

tal que para toda terna de expresiones **M**, **N** y **P**

fst3 $($**terna M N P**$) \rightarrow$ **M**
snd3 $($**terna M N P**$) \rightarrow$ **N**
thd3 $($**terna M N P**$) \rightarrow$ **P**

  - ¿Qué se podría usar para representar?
    - ¡Varios booleanos!

Lambda cálculo, pares

pair $=_{def}$ $(\lambda x\ y.\ \lambda b.$ **if** b **then** x **else** $y)$
fst $=_{def}$ $(\lambda p.\ p$ **True**$)$
snd $=_{def}$ $(\lambda p.\ p$ **False**$)$

cumple que para todo par de expresiones **M** y **N**

- **fst** (**pair M N**) → **M**
- **snd** (**pair M N**) → **N**

- **Pares**: cierre
  - Los pares también sirven para elegir entre dos opciones
    - Pero en este caso se conocen las opciones
    - Y se espera que el usuario indique cuál opción quiere
  - Un dato puede representarse como la transformación de la información de acceso en la información de resultado final (para ese dato)
    - Desde este punto de vista, pares y booleanos son similares (duales)

Listas

Lambda cálculo, listas

- **Listas**: especificación

$$\begin{aligned} \mathbf{Nil} &=_{\mathtt{def}} \ldots \\ \mathbf{Cons} &=_{\mathtt{def}} (\lambda x\ x_s.\ \ldots) \\ \mathbf{foldr} &=_{\mathtt{def}} (\lambda f.\lambda z.\lambda x_s.\ \ldots) \end{aligned}$$

tal que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_S}$

$$\mathbf{foldr\ F\ Z\ Nil} \rightarrow \mathbf{Z}$$

$$\mathbf{foldr\ F\ Z\ (Cons\ X\ X_S)} = \mathbf{F\ X\ (foldr\ F\ Z\ X_S)}$$

Lambda cálculo, listas

Nil $=_{def}$ ...
Cons $=_{def}$ $(\lambda x\ x_s.\ \ldots)$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ \ldots)$

tal que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** $\rightarrow$ **Z**
- **foldr F Z** (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

- La recursión queda expresada por el **foldr**
  - Cualquier cosa que cumpla la especificación, sirve
  - Hay infinitas soluciones
    - Pero se verá solamente la forma canónica de acá en más
  - La idea es proceder como con los booleanos
    - Que la función de acceso despache la información
    - Que los datos sean las funciones que trabajan
    - Para eso deben invertirse los argumentos de **foldr**

Lambda cálculo, listas

Nil $=_{def}$ ...
Cons $=_{def}$ $(\lambda x\ x_s.\ \ldots)$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ \ldots)$

tal que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** → **Z**
- **foldr F Z (Cons X** $\mathbf{X_s}$**)** = **F X (foldr F Z** $\mathbf{X_s}$**)**

- **Listas**: deducir desde la especificación

foldr F Z Nil → **Z**
foldr F Z (Cons X $\mathbf{X_s}$**)** = **F X (foldr F Z** $\mathbf{X_s}$**)**

- Primero se invierten los argumentos

foldr' Nil F Z → **Z**
foldr' (Cons X $\mathbf{X_s}$**) F Z** = **F X (foldr'** $\mathbf{X_s}$ **F Z)**

- Luego se asocian las funciones de interfaz

(foldr' Nil) F Z → **Z**
(foldr' (Cons X $\mathbf{X_s}$**)) F Z** = **F X (foldr'** $\mathbf{X_s}$ **F Z)**

Lambda cálculo, listas

Nil $=_{def}$ ...
Cons $=_{def}$ $(\lambda x\ x_s.\ \ldots)$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ \ldots)$

tal que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** → **Z**
- **foldr F Z** (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

- **Listas**: deducir desde la especificación, 2

  (foldr' Nil) F Z → **Z**
  (foldr' (Cons X $\mathbf{X_s}$**)) F Z = F X (foldr' X**$_s$ **F Z)**

  - Luego se "pasan" los argumentos como parámetros

    foldr' Nil → $(\lambda f\ z.\ z)$
    foldr' (Cons X $\mathbf{X_s}$**)** = $(\lambda f\ z.\ f$ **X (foldr' X**$_s$ $f\ z))$

  - Finalmente, se decide que **foldr'** sea la identidad

    Nil $=_{def}$ $(\lambda f\ z.\ z)$
    Cons X $\mathbf{X_s}$ $=_{def}$ $(\lambda f\ z.\ f$ **X (**$\mathbf{X_s}$ $f\ z))$

Lambda cálculo, listas

Nil $=_{def}$ ...
Cons $=_{def}$ $(\lambda x\ x_s.\ \ldots)$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ \ldots)$

tal que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** $\rightarrow$ **Z**
- **foldr F Z** (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

- **Listas**: definición

Nil $=_{def}$ $(\lambda f\ z.z)$
Cons $=_{def}$ $(\lambda x\ x_s.\ \lambda f\ z.f\ x\ (x_s\ f\ z))$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ x_s\ f\ z)$

cumple que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

foldr F Z Nil $\rightarrow$ **Z**
foldr F Z (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

Lambda cálculo, listas

Nil $=_{def}$ $(\lambda f\ z.z)$
Cons $=_{def}$ $(\lambda x\ x_s.\lambda f\ z.f\ x\ (x_s\ f\ z))$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ x_s\ f\ z)$

cumplen que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** $\rightarrow$ **Z**
- **foldr F Z** (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

- ¿Cómo se representa la lista **[True,False,True]**?

Cons True (**Cons False** (**Cons True Nil**))
$=_{def}$ $(\lambda x\ x_s.\lambda f\ z.f\ x\ (x_s\ f\ z))$ **True** (**Cons False** (**Cons True Nil**))
$\rightarrow$ $\lambda f\ z.f$ **True** ((**Cons False** (**Cons True Nil**)) $f\ z$)
$=_{def}$ $\lambda f\ z.f$ **True** (($(\lambda x\ x_s.\lambda f'\ z'.f'\ x\ (x_s\ f'\ z'))$ **False** (**Cons True Nil**)) $f\ z$)
$\rightarrow$ $\lambda f\ z.f$ **True** (($\lambda f'\ z'.f'$ **False** ((**Cons True Nil**) $f'\ z'$)) $f\ z$)
$\rightarrow$ $\lambda f\ z.f$ **True** (f **False** ((**Cons True Nil**) $f\ z$))
$\rightarrow$ $\lambda f\ z.f$ **True** (f **False** (f **True** (**Nil** $f\ z$)))
$\rightarrow$ $\lambda f\ z.f$ **True** (f **False** (f **True** z))
$=_{def}$ $\lambda f\ z.f\ \underbrace{(\lambda x\ y.x)}_{\text{True}}\ (f\ \underbrace{(\lambda x'\ y'.y')}_{\text{False}}\ (f\ \underbrace{(\lambda x''\ y''.x'')}_{\text{True}}\ z))$

Código lambda para **[True,False,True]** *"Matrix-like"*

Lambda cálculo, listas

Nil $=_{def}$ $(\lambda f\ z.z)$
Cons $=_{def}$ $(\lambda x\ x_s.\lambda f\ z.f\ x\ (x_s\ f\ z))$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ x_s\ f\ z)$

cumplen que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** → **Z**
- **foldr F Z** (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

- ¿Cómo se representa la lista **`[True,False,True]`**?

  Cons True (**Cons False** (**Cons True Nil**))
  → $\lambda f\ z.f$ **True** (f **False** (f **True** z))

  - Comparar con el resultado del **`foldr`** en Haskell

```
  foldr f z [True,False,True]
= foldr f z (True : False : True : [])
= f True (f False (f True z))
```

  - **En general:** una lista se representa como el resultado de hacer **`foldr`** a esa lista (con parámetros **f** y **z**)

Lambda cálculo, listas

Nil $=_{def}$ $(\lambda f\ z.z)$
Cons $=_{def}$ $(\lambda x\ x_s.\lambda f\ z.f\ x\ (x_s\ f\ z))$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ x_s\ f\ z)$

cumplen que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** → **Z**
- **foldr F Z** (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

- **Listas**: otras operaciones
  - Se definen usando **foldr**
  - Ejemplo: la función **map**

map $=_{def}$ $(\lambda f\ x_s.$ **foldr** $(\lambda x\ z_s.$**cons** $(f\ x)\ z_s)$ **nil** $x_s)$

  - En la versión canónica

map $=_{def}$ $(\lambda f\ x_s.\ x_s\ (\lambda x\ z_s.$**cons** $(f\ x)\ z_s)$ **nil**$)$

Lambda cálculo, listas

Nil $=_{def}$ $(\lambda f\ z.z)$
Cons $=_{def}$ $(\lambda x\ x_s.\lambda f\ z.f\ x\ (x_s\ f\ z))$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ x_s\ f\ z)$

cumplen que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** $\rightarrow$ **Z**
- **foldr F Z** *(***Cons X** $\mathbf{X_s}$*)* = **F X** *(***foldr F Z** $\mathbf{X_s}$*)*

- **Listas**: otras operaciones
  - Ejemplos (para algunas se requieren números y/o bottom):

length $=_{def}$ $(\lambda x_s.\ \ldots)$	**filter** $=_{def}$ $(\lambda x_s.\ \ldots)$	**null** $=_{def}$ $(\lambda x_s.\ \ldots)$
sum $=_{def}$ $(\lambda n_s.\ \ldots)$	**append** $=_{def}$ $(\lambda n_s.\ \ldots)$	**head** $=_{def}$ $(\lambda n_s.\ \ldots)$
prod $=_{def}$ $(\lambda p\ x_s.\ \ldots)$	**recr** $=_{def}$ $(\lambda p\ x_s.\ \ldots)$	**tail** $=_{def}$ $(\lambda p\ x_s.\ \ldots)$

  - **Recordar:** se definen usando **foldr**

Lambda cálculo, listas

Nil $=_{def}$ $(\lambda f\ z.z)$
Cons $=_{def}$ $(\lambda x\ x_s.\lambda f\ z.f\ x\ (x_s\ f\ z))$
foldr $=_{def}$ $(\lambda f.\lambda z.\lambda x_s.\ x_s\ f\ z)$

cumplen que para cualesquiera expresiones **F**, **Z**, **X** y $\mathbf{X_s}$

- **foldr F Z Nil** → **Z**
- **foldr F Z** (**Cons X** $\mathbf{X_s}$) = **F X** (**foldr F Z** $\mathbf{X_s}$)

- **Listas**: cierre
  - Las listas son funciones
    - Ya debería resultar obvio
  - Se representan como el patrón de recursión, *"diferido"*
    - El **foldr** delega el trabajo en los datos
    - Las operaciones se deducen de la especificación
    - Funciona de manera similar al *double dispatch* de OOP
    - OTRA VEZ:
      - un dato puede representarse como la transformación de la información de acceso en la información de resultado final (para ese dato)

Árboles

Lambda cálculo, árboles

- ¿Y otros tipos recursivos?
  - Se representarían en base a su fold, como las listas
  - En la versión canónica, el fold solo delega el trabajo
  - Las operaciones constructoras son consecuencia de la especificación (y la decisión anterior)
  - Ejemplos
    - Árboles binarios
    - Expresiones aritméticas
    - etc.

Lambda cálculo, árboles (ExpA)

- **ExpA**: especificación

Cte	$=_{def}$	$(\lambda n.\ \ldots)$
Suma	$=_{def}$	$(\lambda e_1\ e_2.\ \ldots)$
Prod	$=_{def}$	$(\lambda e_1\ e_2.\ \ldots)$
foldExpA	$=_{def}$	$(\lambda c.\lambda s.\lambda p.\lambda e.\ \ldots)$

tal que para todo par de expresiones **C**, **S**, **P**, **N**, $\mathbf{E_1}$ y $\mathbf{E_2}$

foldExpA C S P (Cte N) → C N

foldExpA C S P (Suma E_1 E_2) = S (foldExpA C S P E_1) (foldExpA C S P E_2)

foldExpA C S P (Prod E_1 E_2) = P (foldExpA C S P E_1) (foldExpA C S P E_2)

Lambda cálculo, árboles

- Se sigue el mismo principio que con las listas
  - Se invierten los argumentos del fold (fold’)
  - Se pasan los argumentos desconocidos como parámetros
  - Se elige que el fold’ sea la identidad
  - Se representan los constructores en base a las funciones resultantes
  - Se completa con el fold realizando el despacho necesario

Lambda cálculo, árboles (ExpA)

- **ExpA**: especificación

Cte	$=_{def}$	$(\lambda n.\lambda c\ s\ p.\ c\ n)$
Suma	$=_{def}$	$(\lambda e_1\ e_2.\lambda c\ s\ p.s\ (e_1\ c\ s\ p)\ (e_2\ c\ s\ p))$
Prod	$=_{def}$	$(\lambda e_1\ e_2.\lambda c\ s\ p.p\ (e_1\ c\ s\ p)\ (e_2\ c\ s\ p))$
foldExpA	$=_{def}$	$(\lambda c.\lambda s.\lambda p.\lambda e.e\ c\ s\ p)$

cumple que para todo par de expresiones **C**, **S**, **P**, **N**, $\mathbf{E_1}$ y $\mathbf{E_2}$

foldExpA C S P (Cte N) → C N

foldExpA C S P (Suma E_1 E_2) = S (foldExpA C S P E_1) (foldExpA C S P E_2)

foldExpA C S P (Prod E_1 E_2) = P (foldExpA C S P E_1) (foldExpA C S P E_2)

Lambda cálculo, árboles (ExpA)

- **ExpA**: otras operaciones
  - Se definen usando **foldExpA**
  - Ejemplo: la función **esCte**

$$\mathbf{esCte} =_{\mathbf{def}} \mathbf{foldExpA}\ (\lambda n.\mathbf{True})\ (\lambda e_1\ e_2.\mathbf{False})\ (\lambda e_1\ e_2.\mathbf{False})$$

  - Definir: **esSuma**, **esProd**, **simplExpA**, **evalExpA**
    - Algunas requieren números (que todavía no se vieron)

Lambda cálculo, árboles

- Otros tipos recursivos
  - Representar en lambda cálculo (en forma canónica)
    - El tipo **`Tree`** de los árboles binarios
    - El tipo **`TG`** de la 2da clase de esquemas
- ¿Y para tipos no recursivos?
  - Se sigue el mismo procedimiento
  - Representar
    - **`Maybe`**
    - Diversos tipos enumerativos (**`Color`**, **`Dir`**, etc.)

Números naturales

Lambda cálculo, números naturales

- **Números naturales**: especificación

$$\begin{aligned} \underline{\mathbf{0}} &=_{\mathbf{def}} \ldots \\ \mathbf{succ} &=_{\mathbf{def}} (\lambda n.\ \ldots) \\ \mathbf{foldNat} &=_{\mathbf{def}} (\lambda s.\lambda z.\lambda n.\ \ldots) \end{aligned}$$

tal que para todo par de expresiones **S**, **Z** y para todo número **N**

$$\mathbf{foldNat\ S\ Z\ \underline{0} \rightarrow Z}$$
$$\mathbf{foldNat\ S\ Z\ (succ\ N) = S\ (foldNat\ S\ Z\ N)}$$

Tomar el desvío...

Bottom

Lambda cálculo, bottom

- **Bottom**: especificación

  bottom $=_{\texttt{def}}$ ...

  tal que **bottom** → **...**

  - Una forma de resolverlo es inspirarse en la paradoja de Russell (o paradoja del barbero)

bottom $=_{def}$ ...

tal que

- **bottom** → ...

Lambda cálculo, bottom

- La paradoja del barbero

En un lejano poblado de un antiguo emirato había un barbero llamado As-Samet. Un día el emir ordenó que los barberos solamente afeitaran a aquellas personas que no pudieran afeitarse. Y así mismo impuso la norma de que todo el mundo se afeitase. Cierto día el emir llamó a As-Samet para que lo afeitara y él le contó sus angustias:

—En mi pueblo soy el único barbero. No puedo afeitar al barbero de mi pueblo, ¡que soy yo!, ya que si lo hago, entonces puedo afeitarme por mí mismo, por lo tanto ¡no debería afeitarme! pues desobedecería vuestra orden. Pero, si por el contrario no me afeito, entonces algún barbero debería afeitarme, ¡pero como yo soy el único barbero de allí!, no puedo hacerlo y también así desobedecería a vos mi señor, oh emir de los creyentes, ¡que Allah os tenga en su gloria!

El emir pensó que sus pensamientos eran tan profundos, que lo premió con la mano de la más virtuosa de sus hijas. Así, el barbero As-Samet vivió para siempre feliz y barbón.

bottom $=_{def}$ ...

tal que

- **bottom** → ...

Lambda cálculo, bottom

- La paradoja del barbero
  - El barbero afeita a todos los que no se afeitan a sí mismos, y solamente a ellos
    - El barbero se podría representar como una función

```
(λx.x x)   -- Toma una persona que no se afeita a sí misma
           -- y retorna el resultado de afeitarla
```

      - La aplicación sería “no se afeita a sí mismo”
      - El barbero recibe a alguien que “no se afeita a sí mismo” y lo retorna
  - ¿El barbero se afeita a sí mismo?

bottom $=_{def}$ ...

tal que

- **bottom** → ...

Lambda cálculo, bottom

- **Bottom**: definición

bottom $=_{def}$ $(\lambda x.x\ x)\ (\lambda x.x\ x)$

cumple que **bottom** → ...

bottom $=_{def}$ $(\lambda x.x\ x)\ (\lambda x.x\ x)$

→ $(\lambda x.x\ x)\ (\lambda x.x\ x)$

$=_{def}$ **bottom**

→ ...

bottom $=_{def} (\lambda x.x\ x)\ (\lambda x.x\ x)$

tal que

- **bottom** → ...

Lambda cálculo, bottom

- **Bottom**: consideraciones
  - Se utiliza *auto-aplicación* para esto
    - Es posible porque no hay tipos
    - Complica la semántica (no cualquier función sirve)
      - Solamente esto es tema para un curso entero
  - Hay infinitas expresiones que cumplen esto
  - La no-terminación trae la consideración de órdenes de evaluación (aplicativo vs. normal)
    - Este tema es para otro curso entero

Recursión general

Lambda cálculo, recursión general

- **Recursión general:** se utiliza el siguiente "truco"
  - dada la ecuación recursiva

    `myFun = ... myFun ...`

    definir **myFun** $=_{\texttt{def}}$ **fix** $(\lambda f. \ldots f \ldots)$

    donde los puntos suspensivos rojos son la representación lambda de los negros
  - La expresión **fix** se denomina ***operador de punto fijo***
    - Debe cumplir ciertas propiedades

Lambda cálculo, recursión general

- **Recursión general:** operador de punto fijo (**fix**)

  $\mathbf{fix} =_{\mathtt{def}} \ldots$

  tal que para toda expresión **F**

  $\mathbf{fix}\ \mathbf{F} \rightarrow \mathbf{F}\ (\mathbf{fix}\ \mathbf{F})$

  - Hay infinitas expresiones que cumplen
  - Una posible es

    $\mathbf{fix} =_{\mathtt{def}} (\lambda x\ f.f\ (x\ x\ f))\ (\lambda x\ f.f\ (x\ x\ f))$

    - Por qué funciona es tema del curso entero de semántica

Lambda cálculo, recursión general

- **Recursión general:** ejemplo

```
fact = \n -> if n==0 then 1
                     else n * (fact (n-1))
```

$$\mathbf{fact} =_{\mathbf{def}} \mathbf{fix}\ (\lambda f.\lambda n.\ \mathbf{if\ isZero}\ n\ \mathbf{then}\ \underline{\mathbf{1}}\ \mathbf{else\ mult}\ n\ (f\ (\mathbf{pred}\ n)))$$

- La función **fact** podría definirse usando **recNat** pero se la usa para ilustrar la recursión general

Lambda cálculo, recursión general

- **Recursión general:** ejemplo

fact $=_{def}$ **fix** $(\lambda f.\lambda n.$ **if isZero** n
then $\underline{1}$
else mult n $(f$ **(pred** $n)))$

fact $=_{def}$ $(\lambda x\ f.f\ (x\ x\ f))\ (\lambda x\ f.f\ (x\ x\ f))$
$(\lambda f.\lambda n.\ (n\ (\lambda m.(\lambda x\ y.y))\ (\lambda x\ y.x))$
$(\lambda s\ z.s\ z)$
$(n\ (\lambda i.\ (f\ (n\ (\lambda p.\lambda b.b\ (\lambda s\ z.\ s\ (p\ (\lambda x\ y.x)\ s\ z))\ (p\ (\lambda x\ y.x)))$
$(\lambda b.b\ (\lambda s\ z.z)\ ((\lambda x.x\ x)(\lambda x.x\ x))\ (\lambda x\ y.y))$
$))\ (\lambda n.\lambda s\ z.s\ (n\ s\ z))\ i)\ (\lambda s\ z.z)))$

- Lambda-matrix para **fact**
  - ¿Pueden encontrar algún elemento conocido?

Conclusiones

Lambda cálculo, conclusiones

- Lambda cálculo, lenguaje que solo tiene funciones
  - No hay datos de orden 0 primitivos
- Tiene el mismo poder expresivo que cualquier otro lenguaje de programación
  - Los datos se pueden construir como funciones
    - La función que espera la información de cómo se accederá, y accede al dato representado
  - La recursión estructural juega un rol fundamental
  - **Los parámetros juegan un rol central en programación**


SPOILER
ALERT!

"*David Bowman tuvo el tiempo justo para una frase cortada, que los hombres que esperaban en el Control de Misión, a mil quinientos millones de kilómetros de allí, no habrían de olvidar jamás en el futuro.*

-- ¡El objeto es hueco… y sigue y sigue… y… ¡oh, Dios mío, está lleno de estrellas!

La Puerta de las Estrellas se abrió. La Puerta de las Estrellas se cerró."

2001 Una odisea espacial

Arthur C. Clarke

Resumen

Resumen

- Se definió un lenguaje con solo funciones
  - Lambda cálculo
- Se representaron diversos elementos tradicionales de programación mediante funciones
  - Booleanos, Pares, Listas, Árboles, Números
  - Bottom, Recursión general

"*Luego esperó, poniendo en orden sus pensamientos y meditando sobre sus poderes aún no probados. Porque aunque era el amo del mundo, no estaba muy seguro qué hacer a continuación.*

Mas ya pensaría en algo."

2001. Una odisea espacial

Arthur C. Clarke


That's all Folks!

Guión, dirección y producción general
Pablo E. -Fidel- Martínez López

Instructores
Federico Sawady O'Connor
Cristian Sottile

Colaboradores
Estefanía Prieto
José Luis Cassano

La participación especial de
Aylen C. Martínez López

Con el apoyo de
Julia Troilo
Lautaro G. Martínez López
Valeria Sawady
Fernando Schapachnik
Fundación Sadosky

Mascotas de la producción
Trece
Fiona

2020
Una materia

Universidad Nacional de Quilmes

UNIVERSIDAD NACIONAL DE LA PLATA

ITBA Instituto Tecnológico de Buenos Aires