

Programación para la Inteligencia Artificial

Programación Funcional

Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
*Campus Sur, Calle Paseo Lote II, Sección Segunda No 112,
Nuevo Xalapa, Xalapa, Ver., México 91097*
mailto:aguerra@uv.mx
<https://www.uv.mx/personal/aguerra/pia>

Maestría en Inteligencia Artificial 2023



Universidad Veracruzana

Justificación

- ▶ Modelo preferido de computabilidad **máquina de Turing** \neq algoritmos en lenguajes de alto nivel.
- ▶ Modelo **AL** [5], un lenguaje tipificado para expresar de manera precisa problemas computacionales.
- ▶ **Completo**, en el sentido de que cualquier computación intuitiva, pueda ser especificada por medios finitos.
- ▶ Puente con el **cálculo- λ** , una teoría de las **funciones computables**



Valores, Expresiones y Transformaciones

- ▶ AL es un lenguaje orientado a **expresiones**.
- ▶ Cómputo: Algoritmos = Expresiones → **Valores**.
- ▶ Las computaciones se definen mediante un conjunto de **reglas de transformación**, aplicadas sistemáticamente hasta que ya no es posible ninguna transformación.
- ▶ La última transformación regresa el **valor** computado.
- ▶ Los valores tienen una interpretación más general: **no son necesariamente atómicos**.



Notación

- ▶ Denotemos a las expresiones e ó e_i para $i \in \{0, \dots, n\}$.
- ▶ Una **computación** es una secuencia:

$$e_0 \rightarrow e_1 \rightarrow \dots \rightarrow e_i \rightarrow e_{i+1} \rightarrow \dots \rightarrow e_n$$

- ▶ donde e_0 es la expresión inicial y e_n es la expresión terminal, ó el **valor** de e_0 .
- ▶ La transformación de e_i a e_{i+1} se da por la aplicación de una **única** regla de transformación.
- ▶ Si e_n es el valor de e_0 , se dice que ambas expresiones significan lo mismo, que tienen la misma **semántica**.



Hablado de semántica

- ▶ Las reglas de transformación **preservan el significado**, pues obviamente reemplazan iguales por iguales.
- ▶ **Ejemplo:** La evaluación de una expresión aritmética preserva significado:

$$((5 + 3) \times (8/4)) \rightarrow (8 \times (8/4)) \rightarrow (8 \times 2) \rightarrow 16$$

- ▶ Las expresiones en la cadena difieren sintácticamente, pero son equivalentes semánticamente.
- ▶ La **equivalencia semántica**, por otro lado, no nos dice nada sobre lo que **significa** una expresión.



Sintaxis y Semántica

- ▶ En un sentido estricto, hablaremos de figuras sintácticas y transformaciones puramente sintácticas de expresiones, en otras expresiones que consideramos semánticamente equivalentes;
- ▶ Sin que podamos hablar del significado o semántica de las expresiones en si mismas.



Trabajo relacionado

- ▶ Aunque el lenguaje que estamos presentando no es un lenguaje estrictamente funcional, si tiene como implementación más cercana a **Lisp** [6] y a su intérprete EVAL-APPLY.
- ▶ Ofrecer una aproximación formal a la computabilidad cercana a Lisp y otros lenguajes funcionales.



Expresiones atómicas y compuestas

- ▶ Las expresiones simbólicas en AL incluyen **valores constantes**:
 - ▶ Números
 - ▶ Cadenas de caracteres
 - ▶ Valores booleanos
 - ▶ Variables
 - ▶ Operadores aritméticos, lógicos y relacionales.
- ▶ Estas expresiones se llaman **átomos**, puesto que no están compuestos por otras expresiones del lenguaje.



Aplicaciones

- ▶ $(e_0 e_1 \dots e_n)$ denota la **aplicación** de una expresión e_0 a las expresiones $e_1 \dots e_n$.
- ▶ La expresión e_0 está en la posición del **operador** y las otras expresiones en la posición de los **operandos**.
- ▶ Las aplicaciones son las expresiones más importantes del lenguaje, ya que son a ellas que las **reglas de transformación** se aplican.
- ▶ Observen el uso de la notación prefija con fines de uniformidad.



Condicionales

- ▶ `if e_0 then e_1 else e_2` es una **forma sintáctica especial** que denota la aplicación del predicado e_0 a las expresiones consecuente e_1 y alternativa e_2 .
- ▶ Su objetivo es **elegir** e_1 o e_2 para su posterior evaluación, dependiendo del valor del predicado e_0 (`true` o `false`).



Abstracción funcional

- ▶ $\lambda v_1 \dots v_n$ in e_0 denota una **abstracción** de las variables $v_1 \dots v_n$ de la expresión e_0 ; o una función sin nombre de n **parámetros formales** $v_1 \dots v_n$ cuyo **cuerpo** e_0 especifica la computación de los valores funcionales.
- ▶ Se dice que λ **liga** las variables $v_1 \dots v_n$ en el cuerpo e_0 que determina su **alcance** en lo que se conoce como **abstracción lambda**.



Listas

- ▶ $\langle e_1 \dots e_n \rangle$ denota una **lista** n-aria ó una secuencia de expresiones en un orden particular, en el que tiene sentido hablar del primer, último e i -ésimo elemento.
- ▶ La **lista vacía** se denota por $\langle \rangle$.



Variables locales recursivas

- ▶ letrec $f_1 = e_1 \dots f_k = e_k$ in e_0 define un conjunto de **ecuaciones** que igualan las variables f_1, \dots, f_k con las expresiones e_1, \dots, e_k recursivamente y las ligan en las expresiones e_1, \dots, e_k y también en e_0 .
- ▶ Las variables pueden verse como **nombres** de funciones, para hacer referencia a la parte derecha de la ecuación más adelante en la expresión letrec.
- ▶ El valor de la expresión es el valor de la **expresión meta** e_0 en la cual la ocurrencia de los identificadores f_1, \dots, f_k es remplazada recursivamente por el lado derecho de sus definiciones.



Variables locales no recursivas

- ▶ $\text{let } v_1 = e_1 \dots v_n = e_n \text{ in } e_0$ es una versión no recursiva de `letrec`.
- ▶ **Ninguna** otra expresión es válida en el lenguaje.



Formas sintácticas

- ▶ De manera concisa las **formas sintácticas** pueden escribirse como:

$$\begin{aligned}
 e =_s & \text{ const } | \text{ var } | \text{ oper } | \\
 & (e_0 e_1 \dots e_n) | \\
 & \text{if } e_0 \text{ then } e_1 \text{ else } e_2 | \\
 & \text{let } v_1 = e_1 \dots v_n = e_n \text{ in } e_0 | \\
 & \text{lambda } v_1 \dots v_n \text{ in } e_0 | \\
 & \langle \rangle | \langle e_1 \dots e_n \rangle | \\
 & \text{letrec } f_1 = e_1 \dots f_n = e_n \text{ in } e_0.
 \end{aligned}$$

donde el signo $=_s$ denota **equivalencia sintáctica**.



Ejemplos de programas válidos en AL

▶ Factorial:

$$\text{fac} = \text{lambda } n \text{ in if } (> n 1) \text{ then } (* n (\text{fac}(- n 1))) \text{ else } 1)$$

▶ Ejemplo de segundo orden:

$$\text{doble} = \text{lambda } f \ u \text{ in } f(f(u))$$
$$\text{cuadrado} = \text{lambda } x \text{ in } (* x x)$$

▶ Intuitivamente $\text{fac } 5 \Rightarrow 120$ y $\text{doble cuadrado } 2 \Rightarrow 16$, pero
¿Cómo sucede ese cómputo?



Evaluador

- ▶ Definiremos un **evaluador abstracto** llamado EVAL.
- ▶ Se le debe considerar una **meta-función** que mapea cada forma sintáctica en otra que representa su valor.
- ▶ En este sentido, EVAL puede considerarse la **función semántica** del lenguaje.
- ▶ Este meta-función se define mediante aplicaciones **recursivas** sobre todas las sub-expresiones, o algunas de ellas seleccionadas, de las **formas sintácticas** del lenguaje.



Estrategia de evaluación

- ▶ La selección de las sub-expresiones estará determinada por una estrategia de evaluación que debería computar valores resultantes con un casi **mínimo costo computacional**.
- ▶ Los operandos de una aplicación generalmente deberán evaluarse **antes** que el operador se aplique a ellos, i.e., **llamada por valor**.



Evaluación de expresiones

- ▶ Las **expresiones atómicas** evalúan a si mismas:

$$\text{EVAL}[\![e]\!] = e$$

t.q. $e =_s \text{const} \mid \text{var} \mid \text{oper.}$

- ▶ Para aplicaciones con **expresiones compuestas**:

$$\text{EVAL}[\![(e_0 e_1 \dots e_n)]\!] = \text{EVAL}[\![(\text{EVAL}[\![e_0]\!] \text{EVAL}[\![e_1]\!] \dots \text{EVAL}[\![e_n]\!])]\!]$$



Evaluación de los condicionales

- ▶ Para los **condicionales** tenemos:

$$\text{EVAL}[\text{if } e_0 \text{ then } e_1 \text{ else } e_2] = \begin{cases} \text{EVAL}[e_1] & \text{Si } \text{EVAL}[e_0] = \textit{true} \\ \text{EVAL}[e_2] & \text{Si } \text{EVAL}[e_0] = \textit{false} \\ \text{if } \text{EVAL}[e_0] \text{ then } e_1 \text{ else } e_2 & \text{En cualquier otro caso} \end{cases}$$

- ▶ Observen que si la forma e_0 no es un **predicado**, la expresión queda casi idéntica, excepto que la expresión e_0 es evaluada.



Evaluación de variables locales

- ▶ Para las expresiones `let` tenemos:

$$\begin{aligned} \text{EVAL} \llbracket \text{let } v_1 = e_1 \dots v_n = e_n \text{ in } e_0 \rrbracket = \\ \text{EVAL} \llbracket e_0 [v_1 \leftarrow \text{EVAL} \llbracket e_1 \rrbracket \dots v_n \leftarrow \text{EVAL} \llbracket e_n \rrbracket] \rrbracket \end{aligned}$$

- ▶ La forma evalúa al valor de e_0 en donde todas las ocurrencias de las variables ligadas por `let` han sido **substituidas** por sus valores respectivos.
- ▶ Cada **substitución** se denota como $v_i \leftarrow \text{EVAL} \llbracket e_i \rrbracket$ para $i \in \{1, \dots, n\}$.



Evaluación de abstracciones funcionales

- ▶ Para las **abstracciones anónimas** que ocurren en una posición distinta a la del operador, tenemos que:

$$\text{EVAL}[\lambda v_1 \dots v_n \text{ in } e_0] = \lambda v_1 \dots v_n \text{ in EVAL}[e_0]$$

- ▶ Las abstracciones necesitan tener su **cuerpo evaluado** para determinar sus valores.



Evaluación de listas

- ▶ Para las **listas**:

$$\begin{aligned}\text{EVAL}[\langle \rangle] &= \langle \rangle \text{ y} \\ \text{EVAL}[\langle e_1 \dots e_n \rangle] &= \langle \text{EVAL}[e_1] \dots \text{EVAL}[e_n] \rangle\end{aligned}$$

- ▶ se computan recursivamente los valores de sus **sub-expresiones**, preservando su estructura.



Evaluación de variables locales recursivas

- ▶ Para las expresiones letrec:

$$\text{EVAL} \llbracket \text{letrec } \dots f_i = e_i \dots \text{ in } e_0 \rrbracket = \\ \text{EVAL} \llbracket e_0 [\dots f_i \leftarrow \text{EVAL} \llbracket e_i [\dots f_i \leftarrow \text{letrec } \dots \text{ in } f_i \dots] \rrbracket \dots] \rrbracket \rrbracket$$

- ▶ Similar al caso de las variables locales, salvo que las expresiones e_i pueden:
 - ▶ Ser recursivas, hacer referencia a f_i .
 - ▶ Hacer referencia a f_j con $j \neq i$.



Evaluación de variables locales recursivas (2)

- ▶ Estos valores a su vez deben computarse substituyendo nuevamente las ocurrencias de los identificadores f_i en las expresiones, por copias de las expresiones completas en `letrec`, las cuales sin embargo, tienen su **expresión meta** remplazada por los mismos f_i .
- ▶ Las formas sintácticas especializada `letrec ... $f_i = e_i$... in f_i` de hecho representa las expresiones e_i en su forma no evaluada. Tan pronto como los f_i desaparecen de la expresión (**caso base**), la expresión `letrec` misma **desaparece** pues ya no es necesaria para que la computación continúe.



Completando la semántica

- ▶ Hemos recorrido todas las formas sintáctica, pero la definición de EVAL está lejos de ser completa.
- ▶ Hasta el momento sólo hemos cubierto los casos generales de las aplicaciones que tienen expresiones no específicas en las posiciones de operador y operandos.
- ▶ Lo que falta son los casos especiales donde las expresiones en la posición del operador son (o evalúan a) **abstracciones** y **operaciones primitivas**.



Aplicaciones completas, caso base

- ▶ Estas aplicaciones definen realmente las acciones de **transformación de expresiones**, por ejemplo, los casos estándar de **aplicaciones completas** se transforman con EVAL como sigue:

$$\text{EVAL} \llbracket (\text{lambd}a \ v_1 \dots v_n \ \text{in} \ e_0 \ e_1 \dots e_n) \rrbracket = \\ \text{EVAL} \llbracket e_0 \ [v_1 \leftarrow \text{EVAL} \llbracket e_1 \rrbracket] \dots v_n \leftarrow \text{EVAL} \llbracket e_n \rrbracket] \rrbracket$$



Más argumentos que parámetros formales

- ▶ Si el número de argumentos m es **mayor** que la aridad de la abstracción n , su valor es una nueva aplicación cuyo **operador** es una expresión resultado de la aplicación de la abstracción de aridad n a n argumentos y cuyos **operandos** son los restantes $m - n$ argumentos evaluados:

$$\begin{aligned} \text{EVAL} \llbracket (\text{lambd}a \ v_1 \dots v_n \ \text{in} \ e_0 \ e_1 \dots e_m) \rrbracket \mid m > n = \\ \text{EVAL} \llbracket (\text{EVAL} \llbracket e_0 [v_1 \leftarrow \text{EVAL} \llbracket e_1 \rrbracket \dots v_n \leftarrow \text{EVAL} \llbracket e_n \rrbracket] \rrbracket) \\ \text{EVAL} \llbracket e_{n+1} \rrbracket \dots \text{EVAL} \llbracket e_m \rrbracket) \rrbracket \end{aligned}$$



Aplicaciones parciales

- ▶ Si la aridad n de la abstracción excede el número de argumentos m , en cuyo caso tenemos una **aplicación parcial**, el valor es definido como sigue:

$$\text{EVAL} \llbracket (\text{lambda } v_1 \dots v_n \text{ in } e_0 \ e_1 \dots e_m) \rrbracket \mid m < n = \\ \text{lambda } v_{m+1} \dots v_n \text{ in EVAL} \llbracket e_0 [v_1 \leftarrow \text{EVAL} \llbracket e_1 \rrbracket \dots v_m \leftarrow \text{EVAL} \llbracket e_m \rrbracket] \rrbracket \rrbracket$$



Conflictos entre nombres y Cerraduras

- ▶ El **renombrado de variables** no se considera una solución apropiada dada su complejidad inherente. Podemos adoptar una estrategia conservadora definiendo las evaluaciones parciales como:

$$\text{EVAL} \llbracket (\text{lambda } v_1 \dots v_n \text{ in } e_0 \ e_1 \dots e_m) \rrbracket \mid m < n = \\ \llbracket \text{EVAL} \llbracket e_m \rrbracket \dots \text{EVAL} \llbracket e_1 \rrbracket \text{ lambda } v_1 \dots v_n \text{ in } e_0 \rrbracket$$

- ▶ Una **cerradura**, que tiene los argumentos evaluados y la abstracción empaquetados entre corchetes, **representa** el valor de una nueva abstracción de aridad $n - m$ sin realmente computarlo.
- ▶ La cerradura se evaluará cuando se convierta en una **aplicación completa**.



Operaciones aritméticas

- ▶ Para las **operaciones aritméticas** (binarias) que denotaremos con op_arit , y usando num , num_1 y num_2 para denotar números, tenemos que:

$$\begin{aligned}
 EVAL[(op_arit\ e_1\ e_2)] = & \\
 \left\{ \begin{array}{l} num \\ (op_arit\ EVAL[e_1]\ EVAL[e_2]) \end{array} \right. & \begin{array}{l} \text{Si } num_1 = EVAL[e_1] \wedge num_2 = EVAL[e_2] \\ \text{En cualquier otro caso} \end{array}
 \end{aligned}$$



Predicados

- ▶ Para las **operaciones relacionales**, denotadas por op_rel , tenemos que además, str_1 y str_2 denotan cadenas de caracteres, mientras que $bool$ denota un valor booleano:

$$\text{EVAL}[(op_rel\ e_1\ e_2)] = \begin{cases} bool & \text{Si } str_1 = \text{EVAL}[e_1] \wedge str_2 = \text{EVAL}[e_2] \\ bool & \text{Si } num_1 = \text{EVAL}[e_1] \wedge num_2 = \text{EVAL}[e_2] \\ (op_rel\ \text{EVAL}[e_1]\ \text{EVAL}[e_2]) & \text{En cualquier otro caso} \end{cases}$$



Operaciones sobre listas

$$\text{EVAL} \llbracket (\text{empty } e) \rrbracket = \begin{cases} \text{true} & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle \rangle \\ \text{false} & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle e_1 \dots e_n \rangle \\ (\text{empty } \text{EVAL} \llbracket e \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

$$\text{EVAL} \llbracket (\text{first } e) \rrbracket = \begin{cases} e_1 & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle e_1 \dots e_n \rangle \\ (\text{first } \text{EVAL} \llbracket e \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

$$\text{EVAL} \llbracket (\text{rest } e) \rrbracket = \begin{cases} \langle e_2 \dots e_n \rangle & \text{Si } \text{EVAL} \llbracket e \rrbracket = \langle e_1 e_2 \dots e_n \rangle \\ (\text{rest } \text{EVAL} \llbracket e \rrbracket) & \text{En cualquier otro caso} \end{cases}$$

$$\text{EVAL} \llbracket (\text{append } e_1 e_2) \rrbracket = \begin{cases} \langle e_{11} \dots e_{1n} \dots e_{21} \dots e_{2m} \rangle & \text{Si } \text{EVAL} \llbracket e_1 \rrbracket = \langle e_{11} \dots e_{1n} \rangle \wedge \\ & \text{EVAL} \llbracket e_2 \rrbracket = \langle e_{21} \dots e_{2m} \rangle \\ (\text{append } \text{EVAL} \llbracket e_1 \rrbracket \text{EVAL} \llbracket e_2 \rrbracket) & \text{En cualquier otro caso} \end{cases}$$



Observaciones

- ▶ La definición de EVAL incluye una **verificación dinámica de tipos**. Las aplicaciones de esas funciones son evaluadas solo si sus argumentos son tipo compatibles, en cualquier otro caso quedan casi intactas (sus argumentos son substituidos por sus valores).
- ▶ La meta función EVAL define la **semántica operacional** del lenguaje AL. Nos dice:
 - ▶ que significa una expresión en el lenguaje (su **valor**);
 - ▶ y también cómo una persona o una máquina pueden computar ese valor.



Paralelismo y concurrencia

- ▶ Cuando EVAL aparece en una aplicación, se propaga al frente de sus sub-expresiones, pero el EVAL al frente de la aplicación no desaparece.
- ▶ Esto significa que las sub-expresiones deben ser evaluadas **antes** que la aplicación completa pueda ser evaluada.
- ▶ Sin embargo, puesto que las sub-expresiones son sintácticamente **independientes**, pueden ser evaluadas en cualquier orden, incluso simultáneamente; y sus valores siempre serán los mismos!



Ejemplo de evaluación: factorial

$$\begin{array}{c}
 \text{EVAL}[\text{[} \textit{fac} \text{ 5 } \text{]}] \\
 \downarrow \\
 \text{EVAL}[\text{[EVAL}[\text{[} \textit{fac} \text{]}] \text{ EVAL}[\text{[5]}] \text{]}] \\
 \downarrow \\
 \text{EVAL}[\text{[lambda } n \text{ in if } (> \text{ } n \text{ 1}) \text{ then } (* \text{ } n \text{ (} \textit{fac}(- \text{ } n \text{ 1))) \text{ else 1) 5]}] \\
 \downarrow \\
 \text{if EVAL}[\text{[} (> \text{ 5 1) \text{]}] \text{ then EVAL}[\text{[} (* \text{ 5 (} \textit{fac}(- \text{ 5 1))) \text{]}] \text{ else EVAL}[\text{[1]}] \\
 \downarrow \\
 \text{if } \textit{true} \text{ then EVAL}[\text{[} (* \text{ 5 (} \textit{fac}(- \text{ 5 1))) \text{]}] \text{ else EVAL}[\text{[1]}] \\
 \downarrow \\
 \text{EVAL}[\text{[} (* \text{ 5 (} \textit{fac}(- \text{ 5 1))) \text{]}] \\
 \downarrow \\
 \text{EVAL}[\text{[} (* \text{ 5 EVAL}[\text{[} \textit{fac}(4) \text{]}] \text{]}] \\
 \vdots \\
 \text{EVAL}[\text{[} (* \text{ 5 } (* \text{ 4 } (* \text{ 3 } (* \text{ 2 1)))) \text{]}] \\
 \vdots \\
 120
 \end{array}$$


Referente histórico

- ▶ Leibniz tenía como ideal [1]:
 1. Crear un **lenguaje universal** en el que se pudieran enunciar todos los problemas posibles.
 2. Encontrar un **método de decisión** para resolver los problemas así enunciados.
- ▶ (1) se logra adoptado alguna forma de teoría de conjuntos formulada en lógica de primer orden. (2) Church [3, 4] y Turing [7] demostraron independientemente la inexistencia de tal método.
- ▶ Para ello, Church [2] inventó el sistema formal llamado **Cálculo- λ** .
- ▶ Turing demostró que sus máquinas y el Cálculo- λ eran modelos **fuertemente equivalentes**, en el sentido que definen la misma clase de funciones computables.



Ideas principales I

- ▶ Es el modelo más cercano a los **algoritmos** y su **evaluación** tal y como se definen en AL –la **reducción** por evaluación de las expresiones del lenguaje.
- ▶ Paradigma de todos los lenguajes de **programación funcional** y general siguiendo el teorema de Church-Turing.
- ▶ Una teoría de las **funciones computables** y las propiedades fundamentales de los **operadores**, sus **aplicaciones** a **operandos** y la construcción sistemática de operadores más complejos –**algoritmos**.



Ideas principales II

- ▶ Su núcleo tiene que ver con poco más que la definición de **variables**, su **alcance**, y la **substitución** ordenada de variables por expresiones.
- ▶ Se trata de un **lenguaje cerrado** en el sentido de que su **semántica** puede definirse en base a equivalencia entre expresiones (o términos) del cálculo mismo.



Abstracción funcional

- ▶ Una **función** f de n variables v_1, \dots, v_n se denota como:

$$f = \lambda v_1 \dots v_n. e_0$$

- ▶ El símbolo f denota el **nombre** de la función y puede ser referenciado en cualquier parte.
- ▶ La expresión del lado derecho de la ecuación define una **abstracción** de las **variables** v_1, \dots, v_n en el **cuerpo** de la función e_0 .
- ▶ **Ejemplo.** $\text{suma} = \lambda xy. (+ x y)$



Aplicación de funciones

- ▶ Una **aplicación** de la función f sobre m expresiones argumento e_1, \dots, e_m tiene la forma:

$$(f \ e_1 \ \dots \ e_m) = (\lambda v_1 \ \dots \ v_n. e_0 \ e_1 \ \dots \ e_m)$$

donde $m = n$ no es necesariamente el caso.

- ▶ **Ejemplos:**

- ▶ $(suma \ 3 \ 4) = 7$

- ▶ $(suma \ 1) = \lambda y. (+ \ 1 \ y)$

- ▶ El caso especial de la **aplicación de una abstracción a las variables abstraídas**, nos da como resultado el cuerpo de la abstracción:

$$(\lambda v_1 \ \dots \ v_n. e_0 \ v_1 \ \dots \ v_n) = e_0$$



Variables libres, acotadas y sustituciones

- ▶ Se dice que una abstracción $\lambda v_1.e_0$ **acota** la variable v_1 en e_0 , ssi v_1 ocurre en e_0 . En ese caso se dice que la variable v_1 está **acotada**;
- ▶ Si la variable v_1 es exactamente la variable de la abstracción λ , se dice que esta **ligada**;
- ▶ En cualquier otro caso se dice que la variable se dice **libre**.
- ▶ **Ejemplo.** $\lambda x.(+ x y)$ tiene a x como variable acotada/ligada y a y como variable libre.
- ▶ Una **sustitución** $e_0[v_1/t_1]$ solo afecta a las ocurrencias libres de v_1 en e_0 .
- ▶ **Ejemplo.** $(x \lambda x.x)[x/1] = (1 \lambda x.x)$



Funciones Curryficadas

- ▶ Para mantener el aparato formal simple y conciso, el Cálculo- λ considera, sin pérdida de generalidad, únicamente **abstracciones de una variable**.
- ▶ Las funciones de más de un argumento se obtienen mediante una **iteración de aplicaciones** descubierta por Schönfinkel y Curry:

$$f = \lambda v_1 \dots v_n. e_0 \equiv \lambda v_1. \lambda v_2. \dots \lambda v_n. e_0$$

- ▶ Ejemplo:

$$\textit{suma} = \lambda x. \lambda y. (+ x y)$$



Sintaxis

- ▶ Asumiendo un conjunto infinito de **variables** Var y uno finito, infinito o vacío de **constantes** $Const$.
- ▶ Una **expresión** (fbf) del Cálculo- λ se define como sigue:

$$e =_s v \mid c \mid (e_0 e_1) \mid \lambda v.e_0$$

- ▶ Es decir, una fbf (o **término- λ**) es una variable ($v \in Var$), o una constante ($c \in Const$), o una aplicación, o una abstracción.
- ▶ Solo estas formas son fbfs.
- ▶ Las variables y las constantes se conocen como **átomos** del sistema.
- ▶ En su forma pura, el Cálculo- λ **no define constantes**. En otro caso, se dice **aplicado**.



Observaciones

- ▶ La expresión $(e_0 e_1)$ denota la **aplicación** de e_0 a e_1 .
- ▶ Se dice que e_0 está en la **posición del operador**; y que e_1 está en la del **operando**.
- ▶ Por ello, es común que e_0 y e_1 se identifiquen como la **función** y el **argumento** de la aplicación, lo cual no es totalmente correcto –la sintaxis del Cálculo- λ permite que cualquier expresión esté en cualquiera de las dos posiciones.
- ▶ **Ejemplo.** $(suma\ 1\ 3)$ es una aplicación válida, pero también lo es $(3\ suma)$.



Aplicaciones de abstracciones

- ▶ Estamos interesados particularmente en aplicaciones de la forma

$$(\lambda v. e_0 \ e_1)$$

que tienen una abstracción en la posición del operador y una expresión válida en la posición del operando.

- ▶ Sin operadores primitivos, las abstracciones son las únicas **funciones** en juego.



β -reducción

- ▶ La belleza del Cálculo- λ puro reside en que sólo necesitamos preocuparnos por una sola regla de transformación:

$$\lambda v. e_0 \ e_1 \rightarrow_{\beta} \ e_0[v \leftarrow e_1]$$

- ▶ La regla se conoce como β -reducción o β -contracción y se dice que **simplifica** o **reduce** el β -redex ($\lambda v. e_0 \ e_1$) a su **reductum** o **contractum** $e_0[v \leftarrow e_1]$.
- ▶ Observen la **substitución** inherente a la reducción.



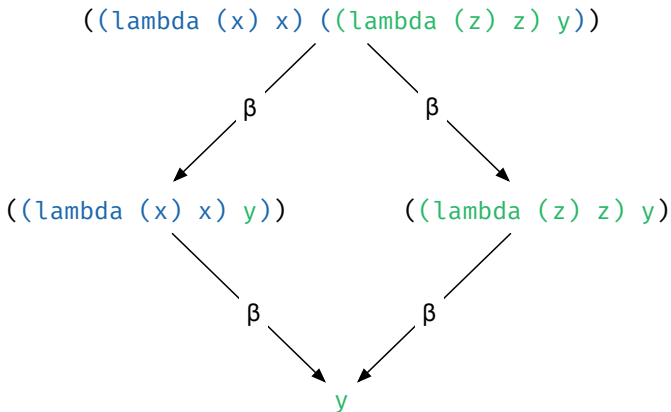
Conflictos entre nombres de variables

- ▶ Desafortunadamente esta regla no es tan simple como parece a primera vista.
- ▶ Existen problemas con respecto a las variables ligadas en las abstracciones y la existencia de variables libres con los **mismos nombres**, en cuyo caso, las **substituciones** no pueden llevarse a cabo sin una acción correctiva.
- ▶ Ejemplo:

$$(\lambda u. \lambda v. u \ w) \rightarrow \lambda v. w \quad \text{y} \quad (\lambda u. \lambda v. u \ v) \rightarrow \lambda v. v$$



No determinismo



Variables libres

- El **conjunto de variables libres** de una expresión e se define por casos sintácticos:

$$FV(e) = \begin{cases} \{v\} & \text{Si } e =_s v \in Var \\ FV(e_0) \cup FV(e_1) & \text{Si } e =_s (e_0 e_1) \\ FV(e_0) \setminus \{v\} & \text{Si } e =_s \lambda v.e_0 \end{cases}$$



Variables acotadas

- ▶ Es posible ofrecer una definición complementaria del **conjunto de variables acotadas** en la expresión e :

$$BV(e) = \begin{cases} \emptyset & \text{Si } e =_s v \in V \\ BV(e_0) \cup BV(e_1) & \text{Si } e =_s (e_0 e_1) \\ BV(e_0) \cup \{v\} & \text{Si } e =_s \lambda v.e_0 \end{cases}$$

- ▶ Una variable v está **libre** en una expresión e , si y sólo si $v \in FV(e)$; y que una variable v está **acotada** en una expresión e , si y sólo si $v \in BV(e)$.



Alcance del abstractor

- ▶ En una abstracción $\lambda v.e$, llamamos al cuerpo e el **alcance** del abstractor λv .
- ▶ Esto significa que todas las ocurrencias libres de v en e están acotadas por λv .
- ▶ **Ejemplo.** En la expresión:

$$\lambda x.x$$

x no es libre, pero en:

$$\lambda x.y$$

y si lo es.



Abstracciones cerradas, abiertas y combinadores

- ▶ Una abstracción cuyo conjunto de variables libres está vacío, se dice **cerrada**, o que es un **combinador**;
- ▶ En otro caso se dice que la abstracción es **abierta**.
- ▶ Los combinadores son de gran relevancia para definir lenguajes basados en Cálculo- λ .
- ▶ **Ejemplo.** Los **combinadores** estándar:

$$I = \lambda x.x$$

$$K = \lambda xy.x$$

$$K_* = \lambda xy.y$$

$$S = \lambda xyz.xz(yz)$$



Substituciones

- Como se lleva a cabo la **substitución** en la β -reducción

$$(\lambda v. e_b \ e_a) \rightarrow_{\beta} e_b[v \leftarrow e_a]$$

$$e_b[v \leftarrow e_a] = \begin{cases} e_a & \text{Si } e_b =_s v \in \text{Var} \\ u & \text{Si } e_b =_s u \in \text{Var} \wedge v \neq_s u \\ (e_0[v \leftarrow e_a] \ e_1[v \leftarrow e_a]) & \text{Si } e_b =_s (e_0 \ e_1) \\ \lambda v. e_0 & \text{Si } e_b =_s \lambda v. e_0 \\ \lambda u. e_0[v \leftarrow e_a] & \text{Si } e_b =_s \lambda u. e_0 \ \wedge \\ & u \notin FV(e_a) \vee v \notin FV(e_b) \\ \lambda w. e_0[u \leftarrow w][v \leftarrow e_a] & \text{Si } e_b =_s \lambda u. e_0 \ \wedge \\ & u \in FV(e_a) \wedge v \in FV(e_b) \ \wedge \\ & w \in \text{Var} \wedge w \notin FV(e_a) \cup FV(e_b) \end{cases}$$



α -conversión

- ▶ La transformación que **renombra** la variable acotada:

$$\lambda u. e_0 \rightarrow_{\alpha} \lambda w. e_0[u \leftarrow w]$$

se conoce como **α -conversión**.

- ▶ Su realización se basa en la aplicación de una función de α -conversión a la abstracción cuya variable acotada necesitamos cambiar:

$$(\lambda v. \lambda w. (v \ w)) \lambda u. e_0$$

- ▶ Esta transformación procede en dos pasos: primero mapea a $\lambda w. (\lambda u. e_0 \ w)$ y después a $\lambda w. e_0[u \leftarrow w]$.



Estructura de ligado

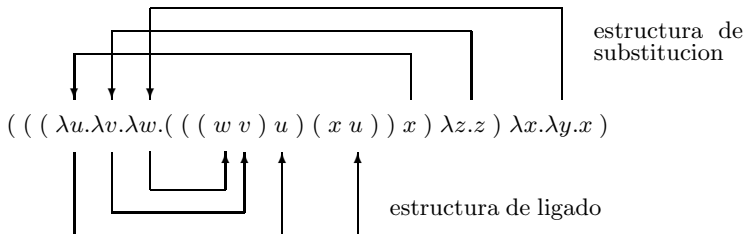
- ▶ La elección de los nombres de las variables ligadas **no es importante**.
- ▶ Su única función es **relacionar** a los abstractores con posiciones sintácticas en el cuerpo de la abstracción.
- ▶ A esta relación se le conoce como **estructura de ligado**.
- ▶ **Ejemplo:**

$$\lambda u. \lambda v. \lambda w. (((w \ v) \ u)(x \ u)) \equiv \lambda x_1. \lambda x_2. \lambda x_3. (((x_3 \ x_2) \ x_1)(x \ x_1))$$



Estructura de sustitución

- ▶ La posición de un abstractor en una secuencia de abstractores también identifica, en el caso de aplicaciones anidadas, el nivel de **anidamiento** en que el argumento será tomado.
- ▶ A esto se le llama **estructura de sustitución**.



Secuencias de reducción

- ▶ Dadas dos λ -expresiones e y e' , se dice que e es β -reducible a e' , denotado por $e \mapsto_{\beta} e'$, si y sólo si e puede ser transformado en e' por una secuencia finita (posiblemente vacía) de β -reducciones y α -conversiones.
- ▶ Esto produce una secuencia e_0, \dots, e_n de λ -expresiones con $e_0 =_s e$ y $e_n =_s e'$ tal que para todos los índices $i \in \{0, \dots, n-1\}$ tenemos que $e_i \rightarrow_{\beta} e_{i+1}$ ó $e_i \rightarrow_{\alpha} e_{i+1}$.



Equivalencia semántica

- ▶ Dos λ -expresiones son **semánticamente equivalentes**, denotado por $e \equiv e'$, si y sólo si e puede ser transformada en e' por una secuencia finita (posiblemente vacía) de β -reducciones, β -reducciones inversas y α -conversiones.
- ▶ Esto es, para todos los índices $i \in \{0, \dots, n-1\}$, debemos tener $e_i \rightarrow_{\beta} e_{i+1}$ ó $e_{i+1} \rightarrow_{\beta} e_i$ ó $e_i \rightarrow_{\alpha} e_{i+1}$.



Forma Normal

- ▶ El objetivo de reducir una expresión- λ e es transformarla en alguna expresión e^{NF} que no contiene más redices, ó a la que no se le puedan aplicar más reglas de β -reducción.
- ▶ Esta expresión se conoce como la **forma normal** ó el valor de la expresión e .
- ▶ Si para llegar de e a e^{NF} necesitamos una secuencia finita de β -reducciones, entonces e^{NF} es también la forma normal de las expresiones- λ **intermedias**.



Indeterminismo

- ▶ Una expresión- λ que contiene diversos redices, permite una elección entre diferentes secuencias **alternativas** de β -reducciones.
- ▶ Podemos alcanzar una forma normal, en:
 - ▶ **todas** las posibles secuencias, p. ej., tras muchas β -reducciones finitas; si tenemos suerte.
 - ▶ **ninguna** de las secuencias posibles, porque no existe una forma normal, i.e., ninguna de las secuencias termina en un número finito de pasos;
 - ▶ **algunas** de las secuencias posibles, pero no en todas, p. ej, algunas secuencias terminan de manera finita, pero otras no.
- ▶ El último caso es muy interesante porque demanda una **estrategia**



Ejemplo del caso 1

- ▶ Consideremos la expresión: $(\lambda u.(\lambda w.(\lambda w.u u) u) w)$ donde:

- ▶ procediendo de afuera hacia adentro:

$$(\lambda u.(\lambda w.(\lambda w.u u) u) w) \rightarrow_{\beta} (\lambda w.(\lambda(w./w /w)w)w) \rightarrow_{\beta} (\lambda w./w w) \rightarrow_{\beta} w$$

- ▶ procediendo de adentro hacia afuera:

$$(\lambda u.(\lambda w.(\lambda w.u u) u) w) \rightarrow_{\beta} (\lambda u.(\lambda w.u u) w) \rightarrow_{\beta} (\lambda u.u w) \rightarrow_{\beta} w$$

- ▶ y aún si comenzásemos por el radice de en enmedio, también llegaríamos a la forma normal w .



Ejemplo del caso 2

- Ahora, una expresión simple que no tiene forma normal es la **auto-aplicación**:

$$(\lambda u.(u u) \lambda u.(u u)) \rightarrow_{\beta} (\lambda u.(u u) \lambda u.(u u)) \rightarrow_{\beta} \dots$$

que se reproduce a si misma.



Ejemplo del caso 3

- ▶ Observen la siguiente aplicación:

$$((\lambda u.\lambda v.u \ \lambda w.w) \ (\lambda u.(u \ u) \ \lambda u.(u \ u)))$$

- ▶ La abstracción $\lambda u.\lambda v.u$ es una **función selector** que reproduce su primer argumento, es decir $\lambda w.w$, pero elimina el segundo, que en este caso particular es la forma auto replicante del ejemplo anterior.
- ▶ De forma que una reducción de adentro hacía afuera lleva a $\lambda w.w$, pero una de afuera hacía adentro no termina.



Reducción en orden normal

- ▶ Una estrategia que garantiza la **terminación**, mediante una **función de transformación** τ_N :

$$\tau_N(e) = \begin{cases} v & \text{Si } e =_s v \in \text{Var} \\ \lambda v. \tau_N(e_b) & \text{Si } e =_s \lambda v. e_b \\ \tau'_N(e) & \text{Si } e =_s (\lambda v. e_b \ e_2) \\ \tau'_N(\tau_N(e_1) e_2) & \text{Si } e =_s (e_1 \ e_2) \text{ y } e_1 \neq_s \lambda v. e_b \end{cases}$$

- ▶ donde:

$$\tau'_N(e) = \begin{cases} \tau_N(e_b[v \leftarrow e_2]) & \text{Si } e =_s (\lambda v. e_b \ e_2) \\ (e_1 \ \tau_N(e_2)) & \text{Si } e =_s (e_1 \ e_2) \text{ y } e_1 \neq_s \lambda v. e_b \end{cases}$$



Evaluador abstracto

- ▶ La función τ_N define un **evaluador abstracto** para expresiones del Cálculo- λ puro, similar al evaluador EVAL de AL.
- ▶ A diferencia de EVAL, τ_N solo se propaga recursivamente a través de los **operadores**, pero no toca los operandos hasta que el operador es procesado y no es una abstracción (el último caso en τ'_N).
- ▶ Si es una abstracción, entonces el operando es substituido por ocurrencias libres de la variable ligada (el primer caso de τ'_N).



Llamadas por nombre y por valor

- ▶ La reducción en orden normal también se conoce como **de afuera a adentro y de izquierda a derecha**.
- ▶ Esta estrategia se conoce también como **llamada por nombre** y es usada por lenguajes de programación como Algol y Simula.
- ▶ Problemas: Reducciones redundantes si el operando es substituído en más de un lugar. Solución: **Evaluación perezosa**.
- ▶ Es posible definir una estrategias como la usada por EVAL, conocida como **primero operandos** o **llamada por valor**.



Teorema de Church-Rosser

- ▶ Sean e_0, e_1, e_2, e_3 λ -expresiones. Entonces $e_0 \mapsto_{\beta} e_1$ y $e_0 \mapsto_{\beta} e_2$ implican que existe una expresión e_3 tal que $e_1 \mapsto_{\beta} e_3$ y $e_2 \mapsto_{\beta} e_3$.
- ▶ La prueba está más allá del contenido de este curso, pero observen que si e_3 es una forma normal, entonces esta forma es **única**.
- ▶ Dicho de otra manera, la forma normal de una expresión- λ es **invariante** con respecto a las diversas secuencias de reducción que se pueden seguir para alcanzarla.



Transparencia referencial

- ▶ Si obviamos el problema del conflicto entre nombres, la reducción- β es bastante sencilla.
- ▶ Lleva a cabo **substituciones libres de contexto** entre iguales; es decir, reemplaza una aplicación por otra, sin causar ningún efecto colateral en la expresión donde se encuentra (su contexto).
- ▶ A esta propiedad se le llama **transparencia referencial**.
- ▶ Además, la sintaxis del Cálculo- λ asegura que los redices- β no se traslapen: No comparten componentes.



Punto fijo

- ▶ El Cálculo- λ no sabe como definir ecuaciones.
- ▶ Podríamos intentar lograr la recursividad vía la auto-aplicación, de la que conocemos el caso especial $(\lambda u.(u u) \lambda u.(u u))$.
- ▶ Mejor aún, deberíamos buscar un **Operador de recursividad** universal, digamos s , tal que: $(s f) = (f(s f))$.
- ▶ Es fácil ver que $(s f)$ es un punto fijo de f : es una expresión que f mapea a si misma. Por lo que necesitamos un **combinador de punto fijo**:

$$Y =_s (p p)$$

donde $p =_s \lambda u.\lambda v.(v((u u) v))$.



Comportamiento del combinador- Y

- Probemos un ejemplo para observar el comportamiento de Y :

$$\begin{aligned}
 (Y f) &=_s ((\lambda u. \lambda v. (v((u u) v))) p) f) \rightarrow_\beta \\
 &(\lambda v. (v ((p p) v))) f \rightarrow_\beta \\
 &(f ((p p) f)) =_s (f Y(f))
 \end{aligned}$$



Ejemplo I

- ▶ Consideren la ecuación para una función recursiva:

$$f = \lambda u. \lambda v. (((f\ u)\ v)((f\ v)\ u))$$

- ▶ Podemos escribirla como una aplicación equivalente:

$$f = (\lambda z. \underbrace{\lambda u. \lambda v. (((z\ u)\ v)((z\ v)\ u))}_{e_b} f)$$

- ▶ Y usando e_b como una abreviatura, como:

$$f = (\lambda z. e_b\ f)$$



Ejemplo II

- ▶ Para obtener la forma de $(Y f) = (f(Y f))$, tenemos:

$$f = (Y \lambda z.e_b)$$

de forma que:

$$(Y \lambda z.e_b) = (\lambda z.e_b (Y \lambda z.e_b))$$



Factorial revisitado

- ▶ Definiciones auxiliares:

$$true \equiv \lambda x. \lambda y. x$$

$$false \equiv \lambda x. \lambda y. y$$

$$cond \equiv \lambda p. \lambda c. \lambda a. ((p\ c)\ a)$$

- ▶ Recursiva:

$$fac \equiv \lambda n. cond\ (= n\ 0)\ 1\ (*(fac\ (-\ n\ 1)))$$

- ▶ Con el combinador-Y:

$$fac \equiv Y(\lambda fac. \lambda n. cond\ (= n\ 0)\ 1\ (*(n\ (fac\ (-\ n\ 1))))$$

donde:

$$(= n\ n) \mapsto true\ y\ (= n\ m) \mapsto false$$



Referencias I

- [1] H Barendregt y E Barendsen. *Introduction to Lambda Calculus*. 2000.
- [2] A Church. "A Set of Postulates for the Foundation of Logic". En: *Annals of Mathematics* 33.2 (1932), págs. 346-366.
- [3] A Church. "A Note on the Entscheidungsproblem". En: *Journal of Symbolic Logic* 1 (1936), págs. 40-41.
- [4] A Church. "An unsolvable problem of elementary number theory". En: *American journal of mathematics* 58.2 (1936), págs. 345-363.
- [5] W Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Berlin, Germany New York: Springer-Verlag, 2005.
- [6] J McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962. ISBN: 0262130114.
- [7] AM Turing. "On the Computable Numbers, with Applications to the Entscheidungsproblem". En: *Proceedings of the London Mathematical Society*. Vol. 42. series 2. 1936, págs. 230-265.

