Práctica de Laboratorio

Intérprete de Lambda Cálculo

Grado en Ingeniería Informática (Q7)
Diseño de Lenguajes de Programación
Curso 2025/26

1 Objetivos

Junto con el enunciado de esta práctica se proporcionan varias implementaciones de un intérprete de lambda cálculo escritas en OCaml, las cuales están inspiradas en las implementaciones que Benjamin C. Pierce cita en su libro *Types and Programming Languages*. Dichas implementaciones procesan expresiones de lambda cálculo escritas ante el prompt de un lazo interactivo, las evalúan y escriben en pantalla los correspondientes resultados. El propósito de esta práctica es estudiar dichas implementaciones y ampliarlas en los términos que se explican a continuación. Se considerarán tanto mejoras en los propios intérpretes, como extensiones en el lenguaje lambda cálculo que reconocen.

Hemos visto tres versiones del intérprete: `lambda-1`, `lambda-2` y `lambda-3`. Las mejoras y extensiones que consideraremos no afectarán en ningún caso a las versiones `lambda-1` y `lambda-2`. En el primer caso, por ser demasiado básica y sobre todo por la incomodidad de manejo que implica el uso de los términos "Church booleans" y "Church numerals" para la representación de los valores lógicos y de los números naturales. Pero la razón principal, en ambos casos, es el hecho de no incluir lambda cálculo tipado. Así pues, afectarán únicamente a la versión más completa: `lambda-3`.

2 Apartados de la práctica

Las mejoras y extensiones propuestas en esta práctica pueden enumerarse bajo la forma de apartados, tal y como se indica a continuación:

1. Mejoras en la introducción y escritura de las lambda expresiones:

   1.1. Reconocimiento de **expresiones multi-línea**. El objetivo es que una expresión se pueda introducir en varias líneas. Y dado que en este caso el salto de línea por sí solo ya no implicará necesariamente el final de una expresión, quizás sea útil introducir algún nuevo símbolo o símbolos para indicar ese aspecto (por ejemplo, un doble punto y coma).

   Esta mejora puede abordarse mediante la manipulación directa del *string* que teclea el usuario cuando introduce un nuevo término, o bien mediante la introducción de nuevos tokens en el analizador léxico y de nuevas reglas gramaticales en el analizador sintáctico (siendo esta última estrategia más elegante).

   Este apartado aporta hasta 0,25 puntos.

   1.2. Implementación de un **"*pretty-printer*" más completo**. Uno de los objetivos aquí es el de intentar minimizar el número de paréntesis a escribir cuando una expresión se transforma en un *string*. Quizás la mejor manera de hacerlo sea mediante una cascada de funciones (guiadas por la gramática del lenguaje), que se van llamando unas a otras según se va profundizando en la estructura interna de la expresión. Y otro aspecto adicional que también puede ser considerado es el de la indentación de los diferentes elementos de las expresiones. Quizás la mejor manera de abordarlo sea mediante el uso del módulo `Format` de OCaml.

   Este apartado aporta hasta 1 punto.

2. Ampliaciones del lenguaje lambda-cálculo:

   2.1. Incorporación de un **combinador de punto fijo interno**, de tal forma que se puedan declarar funciones mediante definiciones recursivas directas. La idea es que en lugar de escribir

```
let fix = lambda f.(lambda x. f (lambda y. x x y)) (lambda x. f (lambda y. x x y)) in
let sumaux =
   lambda f. (lambda n. (lambda m. if (iszero n) then m else succ (f (pred n) m))) in
let sum = fix sumaux in
sum 21 34
```

   podamos escribir

```
letrec sum : Nat -> Nat -> Nat =
   lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m) in
sum 21 34
```

   Con el fin de verificar más exhaustivamente el correcto comportamiento de esta nueva ampliación, escriba en el fichero `examples.txt` proporcionado y en la memoria final de la práctica tres ejemplos adicionales que involucren recursividad múltiple a partir de la suma: una lambda expresión que calcule el producto de dos números naturales (*prod*), otra que calcule el término n-ésimo de la serie de Fibonacci (*fib*), y otra que calcule el factorial de un número natural (*fact*).

   Este apartado aporta hasta 0,75 puntos.

   2.2. Incorporación de un **contexto de definiciones globales**, que permita asociar nombres de variables libres con valores o términos, de forma que éstos puedan ser utilizados en lambda expresiones posteriores. La sintaxis debe ser

```
identificador = término
```

   Por ejemplo:

```
x = true
id = lambda x : Bool. x
```

   De esta forma, si después escribimos `id x`, esta expresión debe ser válida y debe devolver `true`.

   Se recomienda reflexionar sobre cómo debe comportarse esta nueva funcionalidad si al realizar una definición se asocia con un nuevo valor un nombre que ya está en el contexto (es decir, se reutiliza un nombre que ya se utilizó en una definición previa). En función de cómo se maneje este fenómeno, podríamos estar ante un "contexto imperativo" o ante un "contexto funcional". Dado que estamos implementando un intérprete de lambda cálculo, el contexto debe ser funcional. Y para ello existen diferentes alternativas de implementación. Se debe elegir una de ellas, explicando cómo se ha programado y explicando también la razón de dicha elección.

   Y se recomienda que esta nueva funcionalidad permita también realizar definiciones o alias de tipos, lo cual será muy útil para ciertos apartados posteriores. La sintaxis debe ser

```
Identificador = Tipo
```

   Por ejemplo:

```
N = Nat
```

De esta forma, si después escribimos `lambda x : N. x`, esta expresión debe ser válida y debe ser de tipo `Nat -> Nat`, o de tipo `N -> N`, en función de cómo el contexto maneje estas variables de tipo.

Este apartado aporta hasta 1 punto.

2.3. Incorporación del **tipo `String`** para el soporte de cadenas de caracteres, así como de la operación de concatenación de estas cadenas.

Este apartado aporta hasta 0,5 puntos.

2.4. Incorporación de las **tuplas** (para el soporte de los productos cartesianos de cualquier número de elementos de cualquier tipo, incluso de tipos diferentes entre ellos), con las operaciones típicas de proyección basadas en la posición de dichos elementos.

Este apartado aporta hasta 1,25 puntos.

2.5. Incorporación de los **registros** (secuencias finitas de campos de cualquier tipo etiquetados), con las operaciones típicas de proyección basadas en las etiquetas de dichos campos.

Este apartado aporta hasta 1,25 puntos.

2.6. Incorporación de las **variantes**, para poder tratar datos cuya representación se pueda basar en colecciones heterogéneas de valores etiquetados, constituyendo cada una de esas etiquetas un caso diferente de construcción de dichos valores. Considere este ejemplo de ejecución, que incluye la definición de una variante llamada `Int`, de algunos valores de este tipo, y de una función de valor absoluto, cuyo propósito final sería el de incorporar a nuestro intérprete una posible representación de los número enteros y de parte de su aritmética:

```
>> Int = <pos:Nat, zero:Bool, neg:Nat>;;
type Int = <pos : Nat, zero : Bool, neg : Nat>
>> p3 = <pos=3> as Int;;
p3 : <pos : Nat, zero : Bool, neg : Nat> = <pos = 3>
>> z0 = <zero=true> as Int;;
z0 : <pos : Nat, zero : Bool, neg : Nat> = <zero = true>
>> n5 = <neg=5> as Int;;
n5 : <pos : Nat, zero : Bool, neg : Nat> = <neg = 5>
>> abs = L i : Int.
     case i of
       <pos=p> => (<pos=p> as Int)
     | <zero=z> => (<zero=true> as Int)
     | <neg=n> => (<pos=n> as Int);;
abs :
  <pos : Nat, zero : Bool, neg : Nat> -> <pos : Nat, zero : Bool, neg : Nat> = ...
 >> abs p3;;
 - : <pos : Nat, zero : Bool, neg : Nat> = <pos = 3>
 >> abs z0;;
 - : <pos : Nat, zero : Bool, neg : Nat> = <zero = true>
 >> abs n5;;
 - : <pos : Nat, zero : Bool, neg : Nat> = <pos = 5>
```

Una vez incorporadas las variantes, este ejemplo debería funcionar correctamente en su intérprete. Adicionalmente, escriba en el fichero `examples.txt` proporcionado y en la memoria final de la práctica una función recursiva `add` de tipo `Int -> Int -> Int` que implemente la suma de enteros sobre este tipo de dato.

Este apartado aporta hasta 1,5 puntos.

2.7. Incorporación de las **listas** (secuencias finitas de elementos de un mismo tipo), con las operaciones típicas de obtener la cabeza, obtener la cola y consultar si es vacía. Adicionalmente, escriba en el fichero `examples.txt` proporcionado y en la memoria final de la práctica tres lambda expresiones: una que calcule recursivamente en función de la suma la longitud de una lista (*length*), otra que realice la concatenación de dos listas (*append*), y otra que realice la aplicación de una función sobre los elementos de una lista y devuelva la lista de los valores resultantes (*map*). A la hora de resolver este apartado está explícitamente prohibido apoyarse en las listas de OCaml.

Este apartado aporta hasta 1,25 puntos.

2.8. Incorporación de **subtipado**. Más concretamente, se trataría de escribir una función que implemente el polimorfismo de subtipado para registros y funciones (es decir, una función que compruebe si dos tipos dados verifican dicha relación de subtipado), y se trataría también de re-implementar la función general de tipado de forma que use este tipo de polimorfismo allí donde sea aplicable. Piense en al menos dos lambda expresiones que involucren operaciones de subtipado (una para registros, y otra para funciones que traten con registros), y escríbalas también en el fichero `examples.txt` y en la memoria final de la práctica.

Este apartado aporta hasta 0,75 puntos.

3. Redacción de una memoria final:

3.1. Además de entregar el código fuente, debe entregarse también una **memoria** en formato pdf que contenga lo siguiente: un pequeño **manual de usuario** que ilustre (formalmente y con ejemplos de ejecución) las nuevas funcionalidades del intérprete, y también un pequeño **manual técnico** que indique qué módulos de las implementaciones originales han sido modificados y qué tipo de cambios se han realizado sobre ellos para lograr implementar esas nuevas funcionalidades (si bien este segundo manual puede ser sustituido por comentarios en el propio código fuente, siempre y cuando éstos sean suficientemente claros y cumplan la misión anteriormente indicada).

Y dado que dentro de las competencias de esta asignatura figura el manejo de idiomas extranjeros, es obligatorio redactar en inglés no sólo la memoria final, sino también el propio código de las implementaciones, los comentarios de dicho código y los ejemplos de ejecución aportados.

Este apartado aporta hasta 0,5 puntos.

Se recomienda fuertemente visionar el vídeo explicado proporcionado junto con este mismo enunciado, en el cual se presenta cada apartado de la práctica con mucho más detalle y se incluyen muchos más ejemplos de uso.

3 Instrucciones de entrega y evaluación

A continuación se enumeran las instrucciones de entrega y evaluación de esta práctica:

- Respecto a la puntuación de cada apartado, cuando decimos "hasta" nos referimos a que se podrá alcanzar esa cantidad de puntos siempre y cuando los apartados funcionen correctamente y sin anomalías; si este aspecto no se cumple, se aplicarán penalizaciones. Por lo tanto, para obtener la máxima puntuación en esta práctica, es necesario realizar todos los apartados y éstos deben cumplir de manera completa todo lo que se especifica en este enunciado. Y para superar la parte práctica de esta asignatura es necesario obtener 5 o más puntos en esta práctica.

- De cara a la valoración de cada trabajo entregado, también se tendrá en cuenta, entre otros aspectos, la usabilidad del programa, así como la claridad y calidad (y no necesariamente la extensión) de los comentarios del código.

- La **fecha límite de entrega** de la práctica es el viernes 5 de diciembre de 2025. Los trabajos presentados requerirán sus correspondientes **defensas** ante el profesor de prácticas durante las sesiones de los días 9 y 16 de diciembre de 2025.