Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Trimestre Abril-Julio 2025

Etapa IV: Generación de la semántica basada en Lambda Cálculo

Esta es la última fase del proyecto y consiste en traducir un programa escrito en nuestro lenguaje aplicativo en una lambda función que pueda correr en Python. Su entrega consistirá únicamente en implementar un traductor descrito, pero en caso que le sea pasado un programa mal escrito, la salida esperada debe ser idéntica al de la entrega pasada.

El lenguaje del Lambda Cálculo está clasificado dentro de una categoría llamada Lenguajes Aplicativos. Una breve descripción de lo que es un lenguaje aplicativo se hace en la siguiente sección.

1. Lenguajes Aplicativos

Sean V y Σ dos alfabetos. A los símbolos de V se le llamaran símbolos variables o simplemente variables, en cambio a los símbolos de Σ se le llamaran símbolos constante o simplemente constantes. Adicionalmente a dichos alfabetos se usarán los símbolos λ, $'('$, $')'$ y el punto $'.'$. Un lenguaje aplicativo L se define como aquel lenguaje cuyas frases son subconjunto de $(V \cup \Sigma \cup \{\lambda, (,), .\})^*$ cumpliendo con la siguiente definición recursiva:

- Una variable o una constante son frases de L
- Si t_1 y t_2 son frases de L, entonces $t_1(t_2)$ es una frase de L
- Si t_1 es una frase de L, etonces (t_1) es una frase de L
- Si t_1 es una frase de L y x es una variable de V, entonces $\lambda x : t_1$ es una frase de L.

Generalmente un lenguaje aplicativo se usa para representar aplicaciones funcionales. Por ejemplo, la función f aplicada a 2, se escribe como $f(2)$ en lenguaje aplicativo. En el ejemplo anterior f es un símbolo variable, por lo que $f(2)$ indica la evaluación o aplicación de 2 a una función desconocida f. Los símbolos constantes de Σ no se refieren a las constantes como se acostumbran en cálculo, es decir no se refieren solo a un número (como en el caso de la constante 2), sino que también pueden referirse a funciones conocidas. Por ejemplo si *sin* fuese un símbolo constante, entonces este puede ser usado para referirse a la función seno.

La idea detrás del significado de los símbolos de V y Σ, es que los símbolos de V son para referirse a objetos desconocidos, en cambio los símbolos de Σ son para referirse a símbolos conocidos, sean funciones o no. Aunque los símbolos de V se refieren a objetos desconocidos, es posible conocer el tipo de estos símbolos por el contexto. Por ejemplo si x es una variable y t_1 es cualquier término de L, entonces de la frase $x(t_1)$ se puede deducir (aunque no se sepa quién es exactamente x) que x debe ser una función porque se le está aplicando t_1. Cuando un símbolo se refiere a una función, se dice que el símbolo es de tipo $p \to q$, es decir que es una función que recibe un objeto de tipo p y devuelve un objeto de tipo q.

En los lenguajes aplicativos todas las funciones dependen de un solo argumento y no de n, sin embargo una función puede recibir una función o devolver una función. Por ejemplo, es posible que una función sea de tipo $(p \to q) \to r$ o $p \to (q \to r)$, en el primer caso recibiría una función de tipo $p \to q$ y devolvería un objeto de tipo r, y en el otro caso recibiría un objeto de tipo p y devolvería una función de tipo $q \to r$. Si

los enteros se denotan con el tipo t, una forma de codificar la función suma + en un lenguaje aplicativo, es como una función de tipo $t \to (t \to t)$. Es decir, cuando la suma recibe un primer argumento $'a'$, se devuelve una función $+(a)$ que está esperando el segundo argumento para terminar la suma. De esta forma la suma de a y b se escribe en notación aplicativa como $(+(a))(b)$. Es tradición en los lenguajes aplicativos abreviar los paréntesis asumiendo que la apliación funcional asocia a izquierda, es decir que la frase $+(a)(b)$ abrevia $(+(a))(b)$ y nunca $+(a(b))$. En Python el simbolo + es un operador y no puede ser usado en notación aplicativa como en el ejemplo anterior. Esto se debe a que Python soporta el paradigma imperativo y funcional (en lenguaje aplicativo) a la vez, por lo tanto existen síbolos que funcionan para un paradigma y no para otro.

El símbolo λx en un lenguaje aplicativo tiene la intención de referirse a un operador que convierte expresiones en funciones. Es decir, si x es de tipo p, y t_1 es de tipo q, entonces $\lambda x.t_1$ es una función de tipo $p \to q$. Por ejemplo, volviendo al caso de la suma, si los enteros se denotan con el tipo t y x es de tipo t, entonces la expresión $+(x)(2)$ es de tipo t, sin embargo $\lambda x. + (x)(2)$ es de tipo $t \to t$, es decir una función que está esperando el valor entero de la variable x para devolver la suma de x con 2.

2. Algunos combinadores importantes

En el contexto del Lambda Cálculo se le llama combinador a toda función lambda en la que no ocurren constantes y todas sus variables se encuentran abstraidas por un operador λ. Para realizar la traducción correctamente es necesario definir algunos combinadores para que el programa final pueda correr en Python. Por esta razón todo programa que usted traduzca debe empezar con estas definiciones que son las siguientes:

```
Z = lambda g:(lambda x:g(lambda v:x(x)(v)))(lambda x:g(lambda v:x(x)(v)))
true = lambda x:lambda y:x
false = lambda x:lambda y:y
nil = lambda x:true
cons = lambda x:lambda y:lambda f: f(x)(y)
head = lambda p: p(true)
tail = lambda p:p(false)
apply = Z(lambda g:lambda f:lambda x:f if x==nil else (g(f(head(x)))(tail(x))))
lift_do=lambda exp:lambda f:lambda g: lambda x: g(f(x)) if (exp(x)) else x
do=lambda exp:lambda f:Z(lift_do(exp)(f))
```

Cada una de estas definiciones fueron explicadas en clase y en el enunciado del proyecto.

3. Estructura de la traducción

Su traductor debe recibir un archivo "`.imperat`" con el programa a traducir y generar un archivo en Python "`.py`" que contenga el programa traducido usando funciones lambda de Python. El profesor correra su programa de salida en Python para verificar su correcto funcionamiento.

El archivo de salida debe comenzar con todas las líneas de la sección anterior luego contener una asignación de la forma

`program = <traduccionLambda>`,

donde `<traduccionLambda>` es la lambda función resultante de traducir todo el programa de entrada (ver las reglas de traducción en el enunciado del proyecto). Por último, el programa debe terminar con las siguientes dos instrucciones:

```
result = program(<listaValoresPorDefecto>)
print(apply(lambda vn:....lambda v2:lambda v1:{'v1':v1,'v2':v2,....,'vn':vn})(result))
```

donde `<listaValoresPorDefecto>` es la lista de valores por defecto de las variables declaradas en el bloque principal del programa. Esta lista según el enunciado del proyecto es de la forma `cons(dn)(...cons(d2)(cons(d1)(nil))...)` donde `d1`, `d2`,..., `dn` son los valores por defecto de las variables declaradas `v1`, `v2`,..., `vn` en el bloque principal.

La instrucción `print` al final del archivo es para visualizar en formato de diccionario el valor final de todas las variables declaradas en el bloque principal del programa (los identificadores `v1`, `v2`,..., `vn` son los declarados en este bloque). Esta última instrucción es fundamental para facilitar al profesor la corrección del proyecto.

4. Aspectos que no se tomarán en cuenta en la entrega

En la presente etapa del proyecto, por razones de tiempo, no se evaluaran programas imperativos que contengan instrucciones `print`, ni tampoco que contengan escrituras de funciones del tipo `f(x:y)`. Las instrucciones `while` tampoco serán evaluadas, sin embargo usted posee todo lo que se necesita para implementarlas si lo desea, esto es la definición del combinador `do` y la explicación de su uso que se encuentra en el enunciado del proyecto.

5. Ejemplo

Supongamos que se tiene el siguiente programa

```
{
    int a, b, c;
    b := a+b+-c*-a*b-a-b;
    a := 12
}
```

El archivo de salida de su programa debe contener lo siguiente:

```
Z = lambda g:(lambda x:g(lambda v:x(x)(v)))(lambda x:g(lambda v:x(x)(v)))
true = lambda x:lambda y:x
false = lambda x:lambda y:y
nil = lambda x:true
cons = lambda x:lambda y:lambda f: f(x)(y)
head = lambda p: p(true)
tail = lambda p:p(false)
apply = Z(lambda g:lambda f:lambda x:f if x==nil else (g(f(head(x)))(tail(x))))
lift_do=lambda exp:lambda f:lambda g: lambda x: g(f(x)) if (exp(x)) else x
do=lambda exp:lambda f:Z(lift_do(exp)(f))

program = (lambda x1:(apply(lambda x3:lambda x2:lambda x1: cons(x3)(cons(x2)(cons(12)(nil))))))(
(apply(lambda x3:lambda x2:lambda x1: cons(x3)(cons(x1+x2+-x3*-x1*x2-x1-x2)(cons(x1)(nil))))))(x1)))

result = program(cons(0)(cons(0)(cons(0)(nil))))
print(apply(lambda c:lambda b:lambda a:{'a':a,'b':b,'c':c})(result))
```

6. Entrega

La entrega del proyecto es el domingo de la semana 12, antes de las 9:00 am. Su entrega debe incluir lo siguiente:

- Un archivo comprimido `tar.gz` con el código fuente de su proyecto, debidamente documentado. El nombre del archivo deber ser **Etapa4-XX-YY.tar.gz donde XX-YY son los carné de los integrantes del grupo.**

El no cumplimento de los requerimientos podría resultar en el rechazo de su entrega.

Federico Flaviani / fflaviani@usb.ve / Julio 2025