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

Proyecto: Traductor de Lenguaje Imperativo al Lenguaje del Lambda Cálculo

En el presente documento se define un lenguaje de programación imperativo que usa el comando de múltiples guardias, para la instrucción de selección, soporta tipo de datos entero, booleanos y funciones de segmentos de enteros. Se han omitido características normales de otros lenguajes, tales como llamadas a procedimientos, tipos de datos compuestos y números punto flotante, con el objetivo de simplificar su diseño y hacer factible la elaboración de un traductor a lo largo de un trimestre.

El traductor que se debe implementar debe recibir un programa escrito en el lenguaje definido en éste documento y devolver una versión del mismo programa escrito en el lenguaje del Lambda Cálculo. El Lambda Cálculo es una teoría funcional de la Lógica Matemática definida por Alonzo Church, que luego fue utilizada para formalizar la teoría de las funciones computables. El lenguaje del Lambda Cálculo se basa en la aplicación funcional y el operador de abstracción λx. Si f es una función y b es un elemento del dominio de f, entonces la aplicación de la función f al elemento b se denota en el lenguaje del Lambda Cálculo como $f.b$ o $f\ b$. Es decir, lo que en la matemática estándar se escribe como $f(b)$, en el lenguaje del Lambda Cálculo se escribe como $f.b$ o $f\ b$. Por otro lado, el operador de abstracción permite definir una función anónima en base a una expresión. Por ejemplo la diferencia entre $(-b)$ y $(\lambda b. - b)$ en el lenguaje del Lambda Cálculoa, es que la primera es una expresión y la segunda es una función que depende de b. La primera se tiene que entender como un valor fijo $-b$ pero desconocemos b y la segunda como una función que recibe b y devuelve $-b$. El nombre de función anónima se debe a que en la matemática estándar, se necesita un nombre para poder distinguir entre una expresión y una función, es decir que si quisieramos entender a $-b$ como una función, con el lenguaje de las matemáticas tendríamos que buscar un nombre como f para declarar que $f(b) = -b$ y así descartar que $-b$ es una constante. Con el operador de abstracción no hace falta escoger un nombre para la función, ya que por definición $\lambda b. - b$ es una función (que no tiene un nombre).

El lenguaje del Lambda Cálculo se explicará con mayor detalle en la Etapa 4 del proyecto. A continuación se define el lenguaje imperativo que recibirá como entrada el traductor deseado.

1. Estructura de un programa

Un programa tiene la siguiente estructura:

```
{
<declaración de variables>
<instrucción>
}
```

Es decir, una declaración de variables y una instrucción cualquiera (simple o compuesta por varias instrucciones secuenciadas), que se encuentran dentro de un bloque que lo delimita los tokens `{` y `}`. Un programa en GCL podría verse así:

```
{
  int x;
```

```
  x := 1;
  print "Numero: " + x + " ";
  i := 2;
  while i <= 10  -->
      print i + " ";
      i := i+1
  end
}
```

2. Identificadores

Un identificador de variable es una cadena de caracteres de cualquier longitud compuesta únicamente de las letras desde la `A` hasta la `Z` (mayúsculas o minúsculas), los dígitos del 0 al 9, y el caracter `_` . Los identificadores no pueden comenzar por un dígito y son sensibles a mayúsculas; por ejemplo, la variable `var` es distinta a la variable `Var`, las cuales a su vez son distintas a `VAR`. No se exige que sea capaz de reconocer caracteres acentuados ni la ñ.

3. Tipos de datos

Se disponen de 3 tipos de datos en el lenguaje:

- `int`: representan números enteros.
- `bool`: representa un valor booleano, es decir, `true` o `false`.
- `function[..N]`: representa una función donde su dominio es el intervalo entero $[0 \dots N]$ con $N \geq 0$. En la declaración de una función, `N` debe ser un literal entero, es decir no se puede hacer una función , colocando `N` como variable de tipo entera, sino que hay que colocar valores enteros fijos.

Las palabras `int`, `bool` y `function` están reservadas por el lenguaje y se usan en la declaración de tipos de variables.

4. Instrucciones

4.1. Bloque

Permite secuenciar un conjunto de instrucciones y declarar variables locales. Puede usarse en cualquier lugar donde se requiera una instrucción. Su sintaxis es:

```
{
  <declaración de variables>;
  <instrucción 1> ;
  <instrucción 2> ;
  ...
  <instrucción n-1> ;
  <instrucción n>
}
```

El bloque consiste de una sección de declaración de variables, la cual es opcional, y una secuencia de instrucciones separadas por `;`. Nótese que se utiliza el caracter `;` como separador, no como finalizador, por lo que la última instrucción de un bloque no puede terminar con `;`.

La sintaxis de la declaración de variables es:

```
<tipo1> x1, x2, ... , xn1  ;
<tipo2> y1, y2, ... , yn2 ;
...
<tipom> z1, z2, ... ,  zm;
```

Estas variables solo serán visibles a las instrucciones y expresiones del bloque. Se considera un error declarar más de una vez la misma variable en el mismo bloque.

La declaración de variables puede verse como una secuencia de declaraciones de diferentes tipos finalizadas por el caracter `;`. Nótese que por lo tanto la última declaración sí debe terminar con `;`. El tipo de dato a la izquierda de la lista de variables, es el tipo de dato que corresponde a cada variable de la lista.

El bloque de declaración desde el punto de vista semántico define un espacio producto. Concretamente, si $\overline{\texttt{x}}$ es la lista de todas las variables declaradas (en el orden que fueron declaradas en las diferentes líneas) y $T_1, T_2, \ldots, T_n$ es la lista de la semántica de los tipos que corresponde a cada una de las variables anteriores, entonces el bloque de declaración denota el espacio producto

$$\prod_{i=1}^{n} T_i.$$

A este espacio producto se le llama "Espacio de Estados" Esp y a cada tupla de Esp se le llama "Estado".

Por ejemplo la declaración

```
int x, y;
function[..3]  arr ;
bool bo1, bo2;
```

denotaría el espacio de estados $Esp = sem[\![\texttt{int}]\!] \times sem[\![\texttt{int}]\!] \times sem[\![\texttt{function[..3]}]\!] \times sem[\![\texttt{bool}]\!] \times sem[\![\texttt{bool}]\!] = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^{[0..3]} \times \mathbb{B} \times \mathbb{B}$ y un posible estado es el vector $(-1, 0, \{\langle 0, 0\rangle, \langle 1, 3\rangle, \langle 2, 4\rangle, \langle 3, -1\rangle\}, true, false)$. Recuerde que $\mathbb{Z}^{[0..3]}$ denota el conjunto de las funciones con dominio en el intervalo entero [0..3] y rango $\mathbb{Z}$.

Por otro lado. Sea un bloque B_2 que tiene un espacio de estados $\prod_{i=n+1}^{m} T_i$ cuando se calcula de forma aislada y B_1 un bloque con espacio de estados $\prod_{i=1}^{n} T_i$. Se define que cuando B_2 se encuentra anidado dentro de otro bloque B_1, entonces el espacio de estados de B_2 es

$$\prod_{i=1}^{m} T_i.$$

Para efectos de la traducción al lenguaje del Lambda Cálculo, un estado lo implementaremos como una lista en lugar de como un vector. Las listas en el Lambda Cálculo se construyen usando la función *cons*, la cual recibe el primer elemento de la lista f y el resto t de la lista, de modo que $cons\ f\ t$ denota una lista con primer elemento f (first) y cola t (tail). Por ejemplo el estado $(-1, 0, true, false)$ será traducido como la lista $cons\ false\ (cons\ true\ (cons\ 0\ (cons\ -1\ nil)))$, donde nil es la representación de la lista

vacía en el Lambda Cálculo. Note que la lista se construyó con los elementos listados en orden inverso al que aparecen en el vector, esta implementación será ventajosa cuando se desempile un bloque anidado.

Es importante resaltar que los símbolos *cons* y *nil* en este documento son abreviaciones de las funciones anónimas del Lambda Cálculo $\lambda x_1, x_2, x_3.x_3\ x_1\ x_2$ y $\lambda x_1, x_2, x_3.x_2$ respectivamente. También será importante la función anónima $\lambda x_1.x_1$ $(\lambda x_1, x_2.x_2)$ abreviada como *tail*, la cual satisface $tail\ (cons\ f\ t) = t$, es decir, recibe una lista y devuelve la cola de la misma.

Por otro lado las instrucciones se traducen como funciones anónimas que reciben un estado (lista de valores) y devuelven un estado (lista de valores). Por lo tanto, si $T($`<instrucción>`$)$ es la traducción de la instrucción `<instrucción>`, entonces si el bloque

```
{
  <declaración de variables>;
  <instrucción>
}
```

es el bloque de todo el programa, éste se traduce como

$$T(\texttt{<instrucción>})\ (cons\ dx_n\ (cons\ dx_{n-1}\ (\cdots(cons\ dx_1\ nil)))) \ ,$$

donde $x_1, x_2, \ldots, x_n$ es la lista de variables declaradas en el bloque (declaradas en ese orden), $dx_1, dx_2, \ldots, dx_n$ es la lista de los valores por defecto de las variables $x_1, x_2, \ldots, x_n$. Más adelante se explica como traducir un bloque cuando está anidado dentro de otro bloque.

4.2. Skip

La frase `skip` es una instrucción del lenguaje. Dicha inscrucción será traducida al lenguaje del Lambda Cálculo, como la función identidad $\lambda x_1.x_1$.

4.3. Secuenciación

```
<instrucción1> ;
<instrucción2>
```

Una secuenciación de instrucciones es una instrucción que se construye con dos instrucciones `<instrucción1>` y `<instrucción2>`. El operador de secuenciación es el `;`, el cuál es un operador binario, por lo tanto, la última instrucción de varias instrucciones secuenciadas, nunca lleva punto y coma. Si $T($`<instrucción1>`$)$ y $T($`<instrucción2>`$)$ son el resultado de traducir la instrucción 1 y 2 respectivamente, entonces el resultado de traducir la secuenciación es la composición funcional:

$$(\lambda x_1.T(\texttt{<instrucción2>})\ (T(\texttt{<instrucción1>})\ x_1))$$

4.4. Asignación

```
<variable> := <expresión>
```

Ejecutar esta instrucción tiene el efecto de evaluar la expresión del lado derecho y almacenarla en la variable del lado izquierdo. La variable tiene que haber sido declarada previamente y su tipo debe coincidir con el tipo de la expresión, en caso contrario debe arrojarse un error.

Si $\texttt{x}_1, \ldots, \texttt{x}_n$ es la lista de todas las n variables que son visibles dentro de un bloque, listadas en el orden que fueron declaradas y $\texttt{x}_i$ es la i-ésima variable de esa lista, entonces la traducción de la instrucción $\texttt{x}_i$:= `Exp`, cuando se encuentra dentro de ese bloque, es la siguiente:

$$apply\ (\lambda x_n, \ldots x_1.cons\ x_n\ (cons\ x_{n-1}\ (\ldots (cons\ x_{i+1}\ (cons\ Exp\ (\ldots (cons\ x_1\ nil)))))))$$

donde *apply* es una función tal que $apply\ f\ (cons\ x_n\ (cons\ x_{n-1}\ \ldots (cons\ x_1\ nil))) = f\ x_n\ x_{n-1} \ldots x_1$

Más adelante en la sección de expresiones con funciones se explica que si $\texttt{x}_i$ es un `function[..N]`, entonces éste puede inicializarse con una expresión lista de la siguiente forma $\texttt{x}_i$`:= Exp1, Exp2, .., Expn`.

4.5. Salida

```
print <Expresión>
```

Donde `<Expresión>` puede ser una expresión de cualquier tipo o cadenas de caracteres encerradas en comillas dobles, separadas por el operador de concatenación de cadenas (`+`).

Las cadenas de caracteres deben estar encerradas entre comillas dobles (`"`) y solo debe contener caracteres imprimibles. No se permite que tenga saltos de línea, comillas dobles o backslashes (`\`) a menos que sean escapados. Las secuencias de escape correspondientes son `\n`, `\"` y `\\`, respectivamente.

Un ejemplo de `print` válido es el siguiente:

```
print "Hola mundo! \n Esto es una comilla escapada \" y un backslash \\"
```

El operador de concatenación de las cadenas de caracteres es (`+`) el cual convierte automáticamente cualquier argumento entero en cadena de caracteres. Por ejemplo el siguiente `print` imprime `Hola mundo! 1` con un salto de línea al final:

```
print "Hola " + "mundo! " + 1 + "\n"
```

4.6. Condicional y Guardias

```
if <condición1> -->
   <instrucción1>
[] <condición2> -->
   <instrucción2>
 .
 .
[] <condiciónN> -->
   <instrucciónN>
fi
```

La condición debe ser una expresión de tipo `bool`, de lo contrario debe arrojarse un error. Las guardias que tienen las condiciones e instrucciones de la 2 a la N son opcionales.

Ejecutar esta instrucción tiene el efecto de evaluar la condición 1 y si su valor es `true` se ejecuta la instrucción 1; si su valor es `false`, entonces se pasa a evaluar la condición 2 (si existe, sino no se ejecuta ninguna instrucción). Si el valor de la condición 2 es `true`, se ejecuta la instrucción 2; si su valor es `false` se sigue el proceso en la siguiente guardia con la condición 3 y así sucesivamente.

La constante *true* y $false$ se traducen en el Lambda Cálculo como $\lambda x_1, x_2.x_1$ y $\lambda x_1, x_2.x_2$ respectivamente.

La traducción de esta instrucción al lenguaje del Lambda Cálculo es:

$\lambda x_1.(T(\texttt{<condición1>})\ x_1)\ (T(\texttt{<instrucción1>})\ x_1)\ (\ldots ((T(\texttt{<condiciónN>})\ x_1)\ (T(\texttt{<instrucciónN>})\ x_1)\ x_1) \ldots)$

4.7. Iteración

```
while <condición> -->
   <instrucción>
end
```

La condición debe ser una expresión de tipo `bool`. Para ejecutar la instrucción se evalúa la condición, si es igual a `false` termina la iteración; si es `true` se ejecuta la instrucción del cuerpo y se repite el proceso.

La traducción al lenguaje del Lambda Cálculo de esta instrucción es:

$$Y\ (\lambda x_1, x_2.T(\texttt{<condición>})\ x_2\ (x_1\ T(\texttt{<instrucción>}))\ (\lambda x_1.x_1))\ ,$$

donde Y es la función anonima $\lambda x_1.(\lambda x_2.x_1\ (x_2\ x_2))\ (\lambda x_2.x_1\ (x_2\ x_2))$

5. Regla de alcance de variables

Para utilizar una variable primero debe ser declarada al comienzo de un bloque. Es posible anidar bloques entre instrucciones `while` e `if` y también es posible declarar variables con el mismo nombre que otra variable de un bloque exterior. En este caso se dice que la variable interior esconde a la variable exterior y cualquier instrucción del bloque será incapaz de acceder a la variable exterior.

Dada una instrucción o expresión en un punto particular del programa, para determinar si existe una variable y a qué bloque pertenece, el traductor debe partir del bloque más cercano que contenga a la instrucción y revisar las variables que haya declarado, si no la encuentra debe proceder a revisar el siguiente bloque que lo contenga, y así sucesivamente hasta encontrar un acierto o llegar al tope.

El siguiente ejemplo es válido y pone en evidencia las reglas de alcance:

```
{
  int x, y;

  {
    function[..1] x, y;
    x := 1, 2;
    print "print 1 " + x // x será de tipo function
  };

  {
    bool x, y;
    x := true;
    print "print 2 " + x // x será de tipo bool
  };
  x := 1;
  print "print 3 " + x; // x será de tipo int

  while x <= 5 -->
    {
      function[..1] x; // Esconde la declaración de x hecha fuera del while
      x := 4, 5;
      print "print 4 " + x // x será de tipo function
    };
    x := x+1
```

```
  end
}
```

Desde el punto de vista semántico el espacio de estados Esp al inicio del programa anterior viene siendo $\mathbb{Z} \times \mathbb{Z}$, en cambio dentro del primer bloque anidado es $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^{[0..1]} \times \mathbb{Z}^{[0..1]}$. Para un estado fijo en $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^{[0..1]} \times \mathbb{Z}^{[0..1]}$, los cambios en las variables `x`, `y` dentro de ese bloque, afectan la tercera y cuarta coordenada respectivamente del estado. Análogamente, el espacio de estados Esp dentro del segundo bloque anidado es $\mathbb{Z} \times \mathbb{Z} \times \mathbb{B} \times \mathbb{B}$ (la tercera y cuarta coordenada corresponden a los booleanos `x`, `y`) y dentro de la instrucción for es $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}^{[0.,1]}$ (la primera coordenada corresponde al contador del while, que va de 1 al 5, y la tercera coordenada corresponde a la función `x`). La traducción de un estado como listas del Lambda Cálculo deben respetar esta regla, pero listando el vector de estado en orden inverso.

El último aspecto a definir es la traducción de un bloque de código anidado visto como una sola instrucción. Cuando un bloque B_2, cuyas instrucciones son S, está anidado dentro de un bloque B_1, y se tiene que los espacios de estados Esp dentro de B_1 y B_2 son $\prod_{i=1}^{n} T_i$ y $\prod_{i=n+1}^{m} T_i$ respectivamente (con $m \geq n$), entonces si $T(S)$ es la traducción de la instrucción S, se tiene que

$$T(B_2) := \lambda x_1.\overbrace{tail\ \ldots(tail}^{m-n\ veces}\ (T(S)\ (cons\ dx_m\ (\ldots(cons\ dx_{n+1}\ x_1)))))$$

Donde *tail* es la función que devuelve la cola de una lista. Por ejemplo la semántica del segundo bloque anidado del programa anterior, ignorando la instrucción `print "print 2 " . x`, sería:

$$\lambda x_1.tail\ (tail\ (apply\ \lambda x_4, x_3, x_2, x_1.cons\ (x_4\ (cons\ true\ (cons\ x_2\ (cons\ x_1\ nil)))))\ (cons\ false\ (cons\ false\ x_1)))\ ,$$

donde x_2 y x_1 son las posiciones en la lista correspondientes a los enteros y, x fuera del bloque y los $false$ son los valores por defecto de x_4 y x_3 (variables y, x dentro del bloque).

Es importante destacar que esta regla de traducción aplica solo cuando B_2 es un bloque anidado, ya que el caso de la traducción del bloque de todo el programa, ya fue tratado en la sección 4.1.

6. Expresiones

Las expresiones consisten de variables, constantes numéricas, booleanas, y operadores. Al momento de evaluar una variable ésta debe buscarse utilizando las reglas de alcance descritas, y debe haber sido inicializada. Es un error utilizar una variable que no haya sido declarada ni inicializada.

Los operadores poseen reglas de precedencia que determinan el orden de evaluación de una expresión dada. Es posible alterar el orden de evaluación utilizando paréntesis, de la misma manera que se hace en otros lenguajes de programación.

6.1. Expresiones con enteros

Una expresión aritmética estará formada por números naturales (secuencias de dígitos del 0 al 9), variables y operadores aritméticos convencionales. Se considerarán la suma (`+`), la resta (`-`), la multiplicación (`*`), el `-` unario, la inicialización de funciones (`,`) y la modificación de funciones (`:`). Los operadores binarios usarán notación infija y el menos unario usará notación prefija.

La precedencia de los operadores (ordenados comenzando por la menor precedencia) son:

- `,`
- `:`
- `+`, `-` binario
- `*`
- `-` unario

Para los operadores binarios `+`, `-` y `*`, sus operandos deben ser de tipo `int` y su resultado también será de tipo `int`.

La operación inicialización y modificación de funciones (`,` y `:`) se explica en la sección de expresiones con funciones.

6.2. Expresiones con funciones

Una expresión con funciones está formada por números enteros, variables y operadores de inicialización, aplicación y modificación de funciones. Las funciones solo pueden contener elementos de tipos enteros, no existen funciones de booleanos o funciones de funciones en este lenguaje imperativo (aunque en el lenguaje a traducir, que es el Lambda Cálculo sí las hay). Los operadores a considerar son los siguientes:

- inicialización `exp1, exp2, .., expn`: una función `f` con dominio de longitud n se inicializa asignando una lista de enteros separados por comas, de la siguiente manera `f:= exp1, exp2, .., expn`. Es un error asignar una lista de enteros de diferente longitud a la longitud del dominio de la función.
- aplicación funcional `.exp`: la aplicación de una función `f` al entero resultante de la expresión `exp`, se escribe como `f.exp`, en donde `exp` es una expresión entera.
- modificación `exp1:exp2`: una modificación de una función `f` es denotado por la expresión `f(exp1:exp2)`, donde `exp1` y `exp2` son expresiones enteras. El valor de `exp1` corresponde al elemento del dominio cuya imágen se quiere modificar en la función, y el valor de `exp2`, es el valor que tendrá el elemento del dominio `exp1` en la nueva función.

  La expresión `f(exp1:exp2)` se considera como una nueva función, por lo tanto se le puede hacer una nueva modificación o una aplicación funcional con las expresiones `f(exp1:exp2)(exp3:exp4)` y `f(exp1:exp2).exp` respectivamente, y así sucesivamente.

  Una expresión del tipo `f(exp1:exp2)` no genera un cambio persistente en la función `f`, al menos que se haga una asignación de la forma `f:=f(exp1:exp2)`. Se ejemplifica con el siguiente programa:

```
{
  int a;
  function[..2] f;
  f := 3, 2, 1;
  a := f(2:4).2; // a toma el valor de 4 pero no ocurren cambios persistentes en f
  a := f.2;        // a toma el valor de 1
  f := f(2:4);    // f es modificado por la funcion {<0;3>, <1;2>, <2;4>}
  a := f.2;      // a toma el valor de 4
}
```

6.3. Expresiones booleanas

Una expresión booleana estará formada por constantes booleanas (`true` y `false`), variables y operadores booleanos. Se considerarán los operadores `and`, `or` y `!` (and, or y not). También se utilizará notación infija para el `and` y el `or`, y notación prefija para el `!`. Las precedencias son las siguientes (ordenados comenzando por la menor precedencia):

- `or`
- `and`
- `!`

Los operandos de `and`, `or` y `!` deben tener tipo `bool`, y su resultado también será de tipo `bool`.

Además el lenguaje cuenta con predicados capaces de comparar enteros. Los predicados disponibles son menor (`<`), menor o igual (`<=`), igual (`==`), mayor o igual (`>=`), mayor (`>`), y desigual (`<>`). Ambos operandos deben ser del mismo tipo y el resultado será de tipo `bool`.

También es posible comparar expresiones de tipo `bool` utilizando los operadores `==` y `<>`.

Los predicados no son asociativos, a excepción de los operadores `==` y `<>` cuando se comparan expresiones de tipo `bool`.

La precedencia de los operadores relacionales (ordenados comenzando por la menor precedencia) son las siguientes:

- `==`, `<>`
- `<`, `<=`, `>=`, `>`

7. Comentarios y espacios en blanco

Se pueden escribir comentarios de una línea, estilo *C*. Al escribir `//` se ignorarán todos los caracteres hasta el siguiente salto de línea.

El espacio en blanco es ignorado de manera similar a otros lenguajes de programación, es decir, el programador es libre de colocar cualquier cantidad de espacio en blanco entre los elementos sintácticos del lenguaje.

8. Ejemplos

Ejemplo 1:

```
{
  print "Hello world!"
}
```

Ejemplo 2:

```
{
  int count, value, i;
              function[..3] a;
  a := 2, 3, -1, -2;
  count := 3;
  i := 0;
  while i <= count -->
      value := a.i;
      print "Value: " + value;
      i := i+1
  end
}
```

Ejemplo 3:

```
{
  int x;
  x = 0;
  if -5 <= x and x < 0 -->
     print "Del -5 al 0"
  []  x == 0 -->
     print "Tengo un cero"
  []  1 <= x and x < 100 -->
     print "Del 1 al 100"
  fi
}
```