

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS



Lenguajes de Programación

Tarea 9: Inferencia de Tipos, Polimorfismo y Unificación

Johann Ramón Gordillo Guzmán
418046090

José Jhovan Gallardo Valdéz
310192815

Tarea presentada como parte del curso de **Lenguajes de Programación** impartido por la profesora **M.I. Karla Ramírez Pulido**.

05 de Junio del 2020

Link al código fuente: <https://github.com/JohannGordillo/>

1. Preguntas

1. Define los siguientes conceptos con tus propias palabras y en no más de cinco renglones:

a) Variable de tipo.

Respuesta.

Es una variable matemática que se extiende sobre los tipos. Son introducidas por los procedimientos de tipo Λ e inicializadas por las aplicaciones de funciones. Sirven para modelar los tipos de una función.

b) Polimorfismo explícito.

Respuesta.

Es uno de los tipos de polimorfismo, que se da cuando se tiene una función para cada uno de los tipos y número de argumentos que se espera recibir, en las cuales el comportamiento es el mismo y solo cambia el tipo o número de argumentos.

c) Polimorfismo implícito.

Respuesta.

Es uno de los tipos de polimorfismo, que se da cuando no importa el tipo de argumentos que se le pasen a una función, ésta se va a comportar de la misma manera.

2. Para cada una de las siguientes funciones de RACKET, da el tipo de cada una usando variables de tipo. Suponiendo que las listas son estructuras de datos homogéneas.

a) `map`.

Respuesta.

La función `map` aplica el procedimiento dado a los elementos de una lista dada, para regresar una lista con los elementos de la primera lista pero a los que se les ha aplicado el procedimiento.

`map`: $\forall \alpha, \beta. \text{list}(\alpha) \times (\alpha \rightarrow \beta) \rightarrow \text{list}(\beta)$

b) `filter`.

Respuesta.

La función **filter** recibe un predicado y una lista, y regresa los elementos de la lista de entrada que cumplan el predicado.

map: $\forall \alpha. \text{list}(\alpha) \times (\alpha \rightarrow \text{boolean}) \rightarrow \text{list}(\alpha)$

c) foldr.

La función **foldr** aplica un procedimiento dado a los elementos de una lista dada, solo que ha diferencia de la función **map**, foldr combina los valores de retorno en la manera que lo indique el procedimiento. En foldr, la lista se recorre de derecha a izquierda.

Respuesta.

foldr: $\forall \alpha, \beta. \text{list}(\alpha) \times (\alpha \times \beta \rightarrow \beta) \times \beta \rightarrow \beta$

d) foldl.

La función **foldl** aplica un procedimiento dado a los elementos de una lista dada, solo que ha diferencia de la función **map**, foldl combina los valores de retorno en la manera que lo indique el procedimiento. En foldl, la lista se recorre de izquierda a derecha.

Respuesta.

foldl: $\forall \alpha, \beta. \text{list}(\beta) \times (\alpha \times \beta \rightarrow \alpha) \times \alpha \rightarrow \alpha$

3. Para cada una de las siguientes expresiones, realiza su inferencia de tipos generando las restricciones de tipos correspondientes.

a)

```
(define (potencia a b)
  (if (zero? b)
      1
      (* a (potencia a (sub1 b)))))
```

Respuesta.

Primero hay que nombrar las subexpresiones:

1	(if (zero? b) 1 (* a (potencia a (sub1 b)))))
2	(zero? b)
3	1
4	(* a (potencia a (sub1 b)))
5	(potencia a (sub1 b))
6	(sub1 b)

Ahora hay que derivar las subexpresiones. Derivando la primera:

$$\begin{aligned} [[1]] &= [[(\text{if } (\text{zero? } b) \ 1 \ (* \ a \ (\text{potencia } a \ (\text{sub1 } b))))]] \\ &= [[\text{if } [2] \ [3] \ [4]]] \end{aligned}$$

De donde tenemos las siguientes restricciones:

$$\begin{aligned} [[1]] &= [[3]] \\ [[1]] &= [[4]] \\ [[2]] &= \text{boolean} \end{aligned}$$

Derivando la segunda subexpresión:

$$[[2]] = [[(\text{zero? } b)]]$$

De donde tenemos las siguientes restricciones:

$$\begin{aligned} [[(\text{zero? } b)]] &= \text{boolean} \\ [[b]] &= \text{number} \end{aligned}$$

Derivando la tercera subexpresión:

$$[[3]] = [[1]]$$

De donde tenemos la siguiente restricción:

$$[[1]] = \text{number}$$

Derivando la cuarta subexpresión:

$$[[4]] = [[(* \ a \ (\text{potencia } a \ (\text{sub1 } b)))]]$$

De donde tenemos las siguientes restricciones:

$$\begin{aligned} [[(* \ a \ (\text{potencia } a \ (\text{sub1 } b)))] &= \text{number} \\ [[a]] &= \text{number} \\ [[(\text{potencia } a \ (\text{sub1 } b))]] &= \text{number} \end{aligned}$$

Derivando la quinta subexpresión:

$$[[5]] = [[(\text{potencia } a \ (\text{sub1 } b))]]$$

De donde tenemos la siguiente restricción:

$$[[potencia]] = [[a]] \ [[6]] \rightarrow [[(potencia\ a\ (sub1\ b))]]$$

Derivando la sexta subexpresión:

$$[[6]] = [(sub1\ b)]$$

De donde tenemos las siguientes restricciones:

$$\begin{aligned} [(sub1\ b)] &= number \\ [b] &= number \end{aligned}$$

∴ El tipo de *potencia* es $(number\ number \rightarrow number)$ y los tipos de *a* y de *b* son ambos *number*.

b)

```
(define (suma l)
  (if (empty? l)
      0
      (ncons (nfirst l) (suma (nrest l)))))
```

Respuesta.

Primero hay que nombrar las subexpresiones:

<div style="border: 1px solid black; padding: 2px; display: inline-block;">1</div>	<code>(if (empty? l) 0 (ncons (nfirst l) (suma (nrest l))))</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;">2</div>	<code>(empty? l)</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;">3</div>	<code>0</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;">4</div>	<code>(ncons (nfirst l) (suma (nrest l)))</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;">5</div>	<code>(nfirst l)</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;">6</div>	<code>(suma (nrest l))</code>
<div style="border: 1px solid black; padding: 2px; display: inline-block;">7</div>	<code>(nrest l)</code>

Ahora hay que derivar las subexpresiones. Derivando la primera:

$$\begin{aligned} [[1]] &= [[(if (empty? l) 0 (ncons (nfirst l) (suma (nrest l))))]] \\ &= [[if \ 2 \ 3 \ 4]] \end{aligned}$$

De donde tenemos las siguientes restricciones:

```
[[1]] = [[3]]
[[1]] = [[4]]
[[2]] = boolean
```

Derivando la segunda subexpresión:

```
[[2]] = [[(nempty? 1)]]
```

De donde tenemos las siguientes restricciones:

```
[[(nempty? 1)]] = boolean
[[1]] = nlist
```

Derivando la tercera subexpresión:

```
[[3]] = [[0]] = number
```

Derivando la cuarta subexpresión:

```
[[4]] = [[(ncons (nfirst 1) (suma (nrest 1)))]]
```

De donde tenemos las siguientes restricciones:

```
[[(ncons (nfirst 1) (suma (nrest 1)))]] = nlist
[[nfirst 1]] = number
[[suma (nrest 1)]] = nlist
```

Sin embargo, tenemos una contradicción, porque:

```
[[1]] = [[3]] = number
[[1]] = [[4]] = nlist
```

Pero `number` \neq `nlist`.

c) (define (nfilter p l)
 (cond
 [(empty? l) empty]
 [(p (nfirst l)) (ncons (nfirst l) (nfilter p (nrest l)))]
 [else (nfilter p (nrest l))]))

Respuesta.

Primero hay que nombrar las subexpresiones:

1	(cond
	[(empty? l) empty]
	[(p (nfirst l)) (ncons (nfirst l) (nfilter p (nrest l)))]
	[else (nfilter p (nrest l))])
2	(empty? l)
3	empty
4	(p (nfirst l))
5	(nfirst l)
6	(ncons (nfirst l) (nfilter p (nrest l)))
7	(nfirst l)
8	(nfilter p (nrest l))
9	(nrest l)
10	else
11	(nfilter p (nrest l))
12	(nrest l)

Ahora hay que derivar las subexpresiones. Derivando la primera:

$$\begin{aligned}
 [[1]] &= [[(\text{cond } [[2]] \ [[3]] \ [[4]] \ [[6]] \ \text{else } [[8]])]] \\
 &= [[2]] \rightarrow [[3]] \text{ or } [[4]] \rightarrow [[6]] \text{ or } [[10]] \rightarrow [[11]]
 \end{aligned}$$

De donde:

$$\begin{aligned}
 [[2]] &= \text{boolean} \\
 [[4]] &= \text{boolean} \\
 [[3]] &= [[6]] = [[11]]
 \end{aligned}$$

Derivando la segunda subexpresión:

$$[[2]] = [[(\text{empty? l})]]$$

De donde tenemos las siguientes restricciones:

```
[[ (empty? l) ] ] = boolean  
[[ l ] ] = nlist
```

Derivando la tercera subexpresión:

```
[[ [3] ] ] = [[ (empty) ] ] = nlist
```

Derivando la cuarta subexpresión:

```
[[ [4] ] ] = [[ (p (nfirst l)) ] ]
```

De donde obtenemos la siguiente restricción:

```
[[ p ] ] = [[ (nfirst l) ] ] → [[ (p (nfirst l)) ] ]
```

Derivando la quinta subexpresión:

```
[[ [5] ] ] = [[ (nfirst l) ] ]
```

De donde obtenemos las siguientes restricciones:

```
[[ (nfirst l) ] ] = number  
[[ l ] ] = nlist
```

Derivando la sexta subexpresión:

```
[[ [6] ] ] = [[ (ncons (nfirst l) (nfilter p (nrest l))) ] ]
```

De donde obtenemos las siguientes restricciones:

```
[[ (ncons (nfirst l) (nfilter p (nrest l))) ] ] = nlist  
[[ (nfirst l) ] ] = number  
[[ (nfilter p (nrest l)) ] ] = nlist
```


Derivando la séptima subexpresión:

$$[[\boxed{7}]] = [[(\text{nfirst } 1)]]$$

De donde tenemos las siguientes restricciones:

$$\begin{aligned} [[(\text{nfirst } 1)]] &= \text{number} \\ [[1]] &= \text{nlist} \end{aligned}$$

Derivando la octava expresión:

$$[[\boxed{8}]] = [[(\text{nfilter } p \text{ } (\text{nrest } 1))]]$$

De donde obtenemos la siguiente restricción:

$$[[\text{nfilter}]] = [[p]] \text{ } [[(\text{nrest } 1)]] \rightarrow [[(\text{nfilter } p \text{ } (\text{nrest } 1))]]$$

Derivando la novena subexpresión:

$$[[\boxed{9}]] = [[(\text{nrest } 1)]]$$

De donde obtenemos las siguientes restricciones:

$$\begin{aligned} [[(\text{nrest } 1)]] &= \text{nlist} \\ [[1]] &= \text{nlist} \end{aligned}$$

Derivando la décima subexpresión:

$$[[\boxed{10}]] = [[\text{else}]] = [[\text{true}]] = \text{boolean}$$

Derivando la decimoprimera subexpresión:

$$[[\boxed{11}]] = [[(\text{nfilter } p \text{ } (\text{nrest } 1))]]$$

De la que obtenemos la siguiente restricción:

$$[[\text{nfilter}]] = [[p]] \text{ } [[(\text{nrest } 1)]] \rightarrow [[(\text{nfilter } p \text{ } (\text{nrest } 1))]]$$

Derivando la decimosegunda subexpresión:

$$[[\boxed{12}]] = [[(\text{nrest } 1)]]$$

De donde obtenemos las siguientes restricciones:

```
[[nrest 1]] = nlist
[[1]] = nlist
```

∴ La función `nfilter` es de tipo $((\text{number} \rightarrow \text{boolean}) \text{nlist}) \rightarrow \text{nlist}$, y respecto a los argumentos de la función se tiene que `p` es una función de tipo $(\text{number} \rightarrow \text{boolean})$, mientras que `l` es de tipo `nlist`.

4. Usando el Algoritmo de Unificación, muestra la inferencia de tipos de las siguientes expresiones:

a) $(\lambda (x) (x \ 2 \ 3))$

Respuesta.

Primero, nombramos las subexpresiones:

1	$(\lambda (x) (x \ 2 \ 3))$
2	$(x \ 2 \ 3)$
3	2
4	3

El siguiente paso es generar las restricciones de tipo asociadas.

■ Para 1:

$[[\text{1}]] = [[(\lambda (x) (x \ 2 \ 3))]]$

Obtenemos la siguiente restricción:

$[[(\lambda (x) (x \ 2 \ 3))]] = [[x]] \rightarrow [[\text{2}]]$

■ Para 2:

$[[\text{2}]] = [(x \ 2 \ 3)]$

Obtenemos la siguiente restricción:

$[[x]] = [[\text{3}]] \ [[\text{4}]] \rightarrow [(x \ 2 \ 3)]$

■ Para 3:

$[[\text{3}]] = [\text{2}] = \text{number}$

■ Para 4:

$[[\text{4}]] = [\text{3}] = \text{number}$

Finalmente, procedemos a aplicar el Algoritmo de Unificación.

Action	Stack	Substitution
Initialize	$[[1]] = [[x]] \rightarrow [[2]]$ $[[x]] = [[3]] \times [[4]] \rightarrow [[2]]$ $[[3]] = \text{number}$ $[[4]] = \text{number}$	empty
Step 2	$[[x]] = [[3]] \times [[4]] \rightarrow [[2]]$ $[[3]] = \text{number}$ $[[4]] = \text{number}$	$[[1]] \leftarrow [[x]] \rightarrow [[2]]$
Step 2	$[[3]] = \text{number}$ $[[4]] = \text{number}$	$[[1]] \leftarrow ([[3]] \times [[4]] \rightarrow [[2]]) \rightarrow [[2]]$ $[[x]] \leftarrow [[3]] \times [[4]] \rightarrow [[2]]$
Step 2	$[[4]] = \text{number}$	$[[1]] \leftarrow (\text{number} \times [[4]] \rightarrow [[2]]) \rightarrow [[2]]$ $[[x]] \leftarrow \text{number} \times [[4]] \rightarrow [[2]]$ $[[3]] = \text{number}$
Step 2	empty	$[[1]] \leftarrow (\text{number} \times \text{number} \rightarrow [[2]]) \rightarrow [[2]]$ $[[x]] \leftarrow \text{number} \times \text{number} \rightarrow [[2]]$ $[[3]] \leftarrow \text{number}$ $[[4]] \leftarrow \text{number}$

Lo anterior tiene sentido, pues x es un procedimiento que recibe dos valores numéricos y puede regresar un valor numérico, un valor booleano, o incluso algún otro tipo. Es decir, es de tipo $((\text{number} \times \text{number} \rightarrow \text{any}) \rightarrow \text{any})$.

b) $((\lambda (x) (* x 2)) (+ 2 3))$

Respuesta.

Primero, nombramos las subexpresiones:

1	$((\lambda (x) (* x 2)) (+ 2 3))$
2	$(\lambda (x) (* x 2))$
3	$(* x 2)$
4	2
5	$(+ 2 3)$
6	2
7	3

El siguiente paso es generar las restricciones de tipo asociadas.

- Para $\boxed{1}$:

$$[[\boxed{1}]] = [[((\lambda (x) (* x 2)) (+ 2 3))]]$$

Obtenemos la siguiente restricción:

$$[[\boxed{2}]] = [[\boxed{5}]] \rightarrow [[\boxed{1}]]$$

- Para $\boxed{2}$:

$$[[\boxed{2}]] = [[(\lambda (x) (* x 2))]]$$

Obtenemos la siguiente restricción:

$$[[\boxed{2}]] = [[x]] \rightarrow [[\boxed{3}]]$$

- Para $\boxed{3}$:

$$[[\boxed{3}]] = [[(* x 2)]] = [[(* x \boxed{4})]]$$

Obtenemos las siguientes restricciones:

$$[[\boxed{3}]] = \text{number}$$

$$[[\boxed{x}]] = \text{number}$$

$$[[\boxed{4}]] = \text{number}$$

- Para $\boxed{4}$:

$$[[\boxed{4}]] = [[2]] = \text{number}$$

- Para $\boxed{5}$:

$$[[\boxed{5}]] = [[(+ 2 3)]] = [[(+ \boxed{6} \boxed{7})]]$$

De donde obtenemos las siguientes restricciones:

$$[[\boxed{5}]] = \text{number}$$

$$[[\boxed{6}]] = \text{number}$$

$$[[\boxed{7}]] = \text{number}$$

- Para $\boxed{6}$:

$$[[\boxed{6}]] = [[2]] = \text{number}$$

- Para $\boxed{7}$:

$$[[\boxed{7}]] = [[3]] = \text{number}$$

Por lo que las restricciones generadas son:

$$[[\boxed{2}]] = [[\boxed{5}]] \rightarrow [[\boxed{1}]]$$

$$[[\boxed{2}]] = [[x]] \rightarrow [[\boxed{3}]]$$

$$[[\boxed{3}]] = \text{number}$$

$$[[\boxed{x}]] = \text{number}$$

$$[[\boxed{4}]] = \text{number}$$

$$[[\boxed{5}]] = \text{number}$$

$$[[\boxed{6}]] = \text{number}$$

$$[[\boxed{7}]] = \text{number}$$

Finalmente, procedemos a aplicar el Algoritmo de Unificación.

En la siguiente página se muestra una tabla con la aplicación del mismo.

Action	Stack	Substitution
Initialize	$[[2]] = [[5]] \rightarrow [[1]]$ $[[2]] = [[x]] \rightarrow [[3]]$ $[[3]] = \text{number}$ $[[x]] = \text{number}$ $[[4]] = \text{number}$ $[[5]] = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	empty
Step 2	$[[5]] \rightarrow [[1]] = [[x]] \rightarrow [[3]]$ $[[3]] = \text{number}$ $[[x]] = \text{number}$ $[[4]] = \text{number}$ $[[5]] = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow [[5]] \rightarrow [[1]]$
Step 4	$[[5]] = [[x]]$ $[[1]] = [[3]]$ $[[3]] = \text{number}$ $[[x]] = \text{number}$ $[[4]] = \text{number}$ $[[5]] = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow [[5]] \rightarrow [[1]]$
Step 2	$[[1]] = [[3]]$ $[[3]] = \text{number}$ $[[x]] = \text{number}$ $[[4]] = \text{number}$ $[[x]] = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow [[x]] \rightarrow [[1]]$ $[[5]] \leftarrow [[x]]$
Step 2	$[[3]] = \text{number}$ $[[x]] = \text{number}$ $[[4]] = \text{number}$ $[[x]] = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow [[x]] \rightarrow [[3]]$ $[[5]] \leftarrow [[x]]$ $[[1]] \leftarrow [[3]]$
Step 2	$[[x]] = \text{number}$ $[[4]] = \text{number}$ $[[x]] = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow [[x]] \rightarrow \text{number}$ $[[5]] \leftarrow [[x]]$ $[[1]] \leftarrow \text{number}$ $[[3]] \leftarrow \text{number}$
Step 2	$[[4]] = \text{number}$ $\text{number} = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow \text{number} \rightarrow \text{number}$ $[[5]] \leftarrow \text{number}$ $[[1]] \leftarrow \text{number}$ $[[3]] \leftarrow \text{number}$ $[[x]] \leftarrow \text{number}$
Step 2	$\text{number} = \text{number}$ $[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow \text{number} \rightarrow \text{number}$ $[[5]] \leftarrow \text{number}$ $[[1]] \leftarrow \text{number}$ $[[3]] \leftarrow \text{number}$ $[[x]] \leftarrow \text{number}$ $[[4]] \leftarrow \text{number}$

Action	Stack	Substitution
Step 1	$[[6]] = \text{number}$ $[[7]] = \text{number}$	$[[2]] \leftarrow \text{number} \rightarrow \text{number}$ $[[5]] \leftarrow \text{number}$ $[[1]] \leftarrow \text{number}$ $[[3]] \leftarrow \text{number}$ $[[x]] \leftarrow \text{number}$ $[[4]] \leftarrow \text{number}$
Step 2	$[[7]] = \text{number}$	$[[2]] \leftarrow \text{number} \rightarrow \text{number}$ $[[5]] \leftarrow \text{number}$ $[[1]] \leftarrow \text{number}$ $[[3]] \leftarrow \text{number}$ $[[x]] \leftarrow \text{number}$ $[[4]] \leftarrow \text{number}$ $[[6]] \leftarrow \text{number}$
Step 2	empty	$[[2]] \leftarrow \text{number} \rightarrow \text{number}$ $[[5]] \leftarrow \text{number}$ $[[1]] \leftarrow \text{number}$ $[[3]] \leftarrow \text{number}$ $[[x]] \leftarrow \text{number}$ $[[4]] \leftarrow \text{number}$ $[[6]] \leftarrow \text{number}$ $[[7]] \leftarrow \text{number}$

\therefore La función lambda es de tipo $\text{number} \rightarrow \text{number}$.

5. Dada la siguiente base de conocimientos en PROLOG, muestra la unificación de variables asociada a la meta $g(a, X)$.

```
f(a, b).
m(b, c).
g(X, Z) :- f(X, Y), p(Y, Z).
p(X, Y) :- f(X, Y).
p(X, Y) :- m(X, Y).
```

Respuesta.

Primero que nada, propongamos que la meta sea $g(a, W)$. Es equivalente a $g(a, X)$, pero nos evitará confusiones al momento de aplicar Resolución Binaria y posteriormente unificar mediante composición de sustituciones.

Siguiendo el proceso de Resolución Binaria con ayuda del algoritmo de unificación de Martelli-Montanari, tenemos:

1.	$f(a, b)$	\leftarrow		Hip.
2.	$m(b, c)$	\leftarrow		Hip.
3.	$g(X, Z)$	\leftarrow	$f(X, Y), p(Y, Z)$	Hip.
4.	$p(X, Y)$	\leftarrow	$f(X, Y)$	Hip.
5.	$p(X, Y)$	\leftarrow	$m(X, Y)$	Hip.
6.		\leftarrow	$g(a, W)$	Hip.
7.		\leftarrow	$f(a, Y), p(Y, W)$	Res(3, 6, $[X := a, Z := W]$)
8.		\leftarrow	$p(b, W)$	Res(1, 7, $[Y := b]$)
9.		\leftarrow	$m(b, W)$	Res(5, 8, $[X := b, Y := W]$)
10.		\leftarrow	\emptyset	Res(2, 9, $[W := c]$)

\therefore Dado el programa:

$$\mathbb{P} = \{f(a, b) \leftarrow, m(b, c) \leftarrow, g(X, Z) \leftarrow f(X, Y), p(Y, Z), p(X, Y) \leftarrow f(X, Y), p(X, Y) \leftarrow m(X, Y)\}$$

Al agregar la meta $\leftarrow g(a, W)$ al programa, se ha llegado a la cláusula vacía mediante resolución binaria, por lo que $g(a, W)$ es una consecuencia lógica de \mathbb{P} .

Más aún, la composición de las sustituciones es:

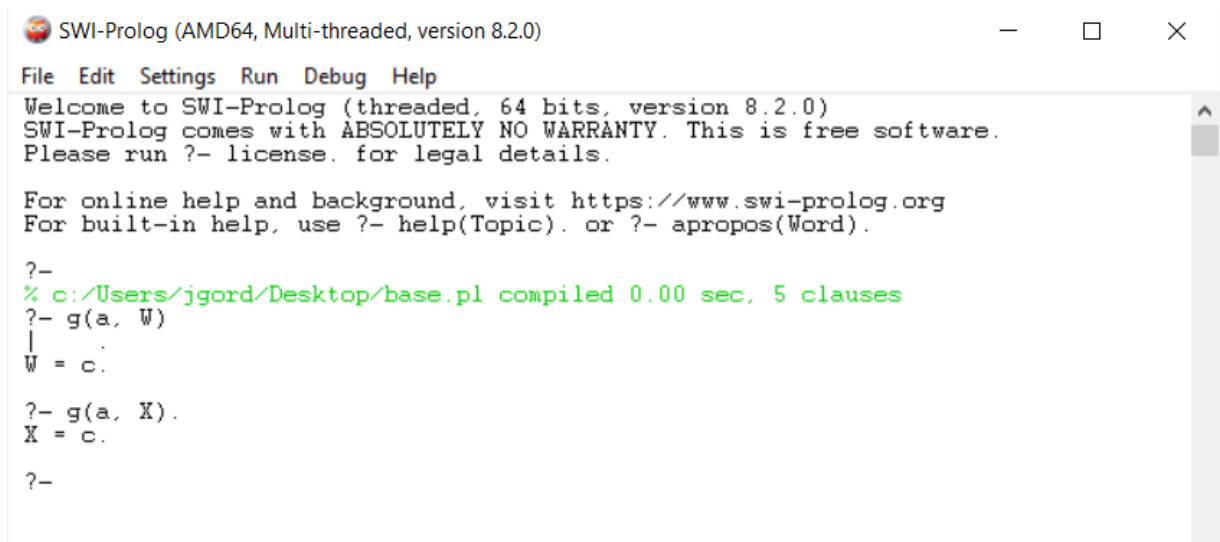
$$\begin{aligned} [X := a, Z := W][Y := b][X := b, Y := W][W := c] &= [X := a, Z := W, Y := b][X := b, Y := W][W := c] \\ &= [X := a, Z := W, Y := b][W := c] \\ &= [X := a, Z := c, Y := b, W := c] \end{aligned}$$

Por lo que $\mu = [X := a, Z := c, Y := b, W := c]$, y al aplicar esta sustitución a la variable W del meta, se obtiene:

$$\begin{aligned} W\mu &= W[X := a, Z := c, Y := b, W := c] \\ &= c \end{aligned}$$

$\therefore W = c$.

Podemos comprobar el resultado anterior usando PROLOG:



```
SWI-Prolog (AMD64, Multi-threaded, version 8.2.0)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.0)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/jgord/Desktop/base.pl compiled 0.00 sec, 5 clauses
?- g(a, W)
|
W = c.

?- g(a, X).
X = c.

?-
```

2. Bibliografía

- Ramírez, K. (2020).
Notas del curso de Lenguajes de Programación.
Facultad de Ciencias - UNAM
Ciudad de México, México.
- Miranda, F. (2020).
Notas del curso de Lógica Computacional.
Facultad de Ciencias - UNAM
Ciudad de México, México.
- Krishnamurthi, S. (2017).
Programming Languages: Application and Interpretation.
Estados Unidos de América.