Part II

Parte II

Functions, Parameters, Scope Rules

- The syntax of a function value (lambda expression) is: ***fn(<type> <symbol>) -> <expr>***
  - The token ***fn*** (a reserved word) can be replaced by ***λ*** (the Zilly Web REPL does it all the time!)
  - ***<type> <symbol>*** describes the ***parameter*** of the function
    - ***<type>*** is the type of the parameter
    - ***<symbol>*** is the name of the parameter
  - ***<expr>*** is the ***body*** of the function (we can also say, the ***body*** of the lambda expression)
  - informally we say "the function *takes* a parameter named ***<symbol>*** of type ***<type>***"
  - note that the function itself (a lambda expression) has no name
  - the ***parameter*** is "bound" (or "initialized") to a value when the function is applied to an ***argument***
    - the preference of saying "bound" or "initialized" depends on the programming paradigm
    - don't worry about which word to choose - we are paradigm independent now, pick one!

Evaluate these expressions in the REPL	Comments
fn(Z n) -> sub(n)(0)	This is a function value: it takes a ***parameter n*** of type ***Z***. Note that the ***parameter n*** plays the role of ***argument*** in the call to function *sub* in the ***body***
(fn(Z n) -> sub(n)(0))(42)	This is a *function application*, a.k.a. *function call*. The function is applied to the ***argument*** *42* Result: the ***body*** of the lambda is evaluated with ***n*** equal to ***42***

Funciones, Parámetros, y Reglas de Alcance

- La sintaxis de un valor de función (expresión lambda) es: ***fn(<tipo> <símbolo>) -> <expr>***
  - El token ***fn*** (una palabra reservada) se puede reemplazar por ***λ*** (¡El Zilly Web REPL lo hace!)
  - ***<tipo> <símbolo>*** describe el ***parámetro*** de la función.
    - ***<tipo>*** es el tipo del parámetro
    - ***<símbolo>*** es el nombre del parámetro.
  - ***<expr>*** es el ***cuerpo*** de la función (también podemos decir, el ***cuerpo*** de la expresión lambda)
  - informalmente, decimos "la función toma un parámetro nombrado ***<símbolo>*** de tipo ***<tipo>***"
  - noten que la función en sí (una expresión lambda) no tiene nombre.
  - el ***parámetro*** se "liga" a (o "inicializa" con) un valor cuando la función se aplica a un ***argumento***.
    - la preferencia entre el uso de "ligar" o "inicializar" depende del paradigma de programación
    - no te preocupes por qué palabra elegir: por ahora somos independientes del paradigma, ¡elige una!

Evalúen estas expresiones en el REPL	Comentarios
fn(Z n) -> sub(n)(0)	Este es un valor de función: toma un ***parámetro n*** de tipo Z. Note que el ***parámetro n*** es un ***argumento*** en la llamada a la función *sub* en el ***cuerpo***.
(fn(Z n) -> sub(n)(0))(42)	Esta es una aplicación de función (llamada a función). La función se aplica al ***argumento*** 42. Resultado: el ***cuerpo*** de la lambda se evalúa con ***n*** igual a ***42***.

Functions, Parameters, Scope Rules

Evaluate these expressions in the REPL	Comments
Z => Z chs := fn(Z n) -> sub(n)(0);	This is a variable definition. The variable, named *chs*, is initialized with the same function we used in the previous example. The type of the variable *chs* is ***Z => Z***, i.e., "function that takes a ***parameter*** of type ***Z*** and returns a result of type ***Z***". That is also **the** type of the initializer**.**
chs(42)	This is a *function application*. We *normally* say "function *chs* is applied to the argument 42" … after all we define variables of type function to give an expressive name (*chs*: change sign) to our functions and avoid having to write a likely long lambda sausage on every application. That's what normal people do ☺

- You may remember the following function from Quiz 1:
  - **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**
- For brevity, let's refer to the lambda expression above as the *lambda-add*
- Note that the body of *lambda-add* is … another lambda expression
  - fn(Z n) -> fn(Z k) -> sub(chs(n))(k)
  - we say that the body of *lambda-add* is a nested lambda expression
  - … and the ***body*** of that nested lambda expression is ***sub(chs(n))(k)***
- We can describe *lambda-add* as "a function that takes a parameter ***n***, of type ***Z***, and …

  … returns a ***function*** that takes a parameter ***k***, of type ***Z***, which returns a result of type ***Z***"
- That is: *lambda-add* is a function that takes a ***Z*** and returns a function of type ***Z => Z***
- *lambda-add* is a function of type ***Z => (Z => Z)***

Funciones, Parámetros, y Reglas de Alcance

Evalúen estas expresiones en el REPL	Comentarios
Z => Z chs := fn(Z n) -> sub(n)(0);	Esta es una definición de variable. La variable, llamada *chs*, se inicializa con la misma función usada anteriormente. El tipo de la variable *chs* es ***Z => Z***, es decir, «función que toma un ***parámetro*** de tipo ***Z*** y retorna un resultado de tipo ***Z***». Ese es también el tipo del inicializador.
chs(42)	Esta es una aplicación de función. Normalmente decimos "*chs* se aplica al argumento 42"... después de todo, definimos variables de tipo función para dar un nombre expresivo (*chs*: cambio de signo) a nuestras funciones y evitar tener que escribir una lambda en cada aplicación. Eso es lo que hace la gente normal. ☺

- Posiblemente recuerdan la siguiente función del Quiz 1:
  - **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**
- Para abreviar, nos referiremos a la expresión lambda anterior como *lambda-add*
- Noten que el cuerpo de *lambda-add* es … otra expresión lambda
  - fn(Z n) -> fn(Z k) -> sub(chs(n))(k)
  - Decimos que el cuerpo de *lambda-add* es una expresión lambda anidada
  - … y el ***cuerpo*** de dicha expresión lambda anidada es ***sub(chs(n))(k)***
- Podemos describir lambda-add como "una función que toma un parámetro ***n***, de tipo ***Z***, y …

  … retorna una función que toma un parámetro ***k***, de tipo ***Z***, que retorna un resultado de tipo ***Z***"
- Es decir: lambda-add es una función que toma ***Z*** y retorna una función de tipo ***Z => Z***
- *lambda-add* es una función de tipo ***Z => (Z => Z)***

Functions, Parameters, Scope Rules

Recap: load this code in the Zilly Editor and run

```
sys.reset();

fn(Z n) -> sub(n)(0)
(fn(Z n) -> sub(n)(0))(18)

Z => Z chs := fn(Z n) -> sub(n)(0);
chs(18)

fn(Z n) -> fn(Z k) -> sub(chs(n))(k)
(fn(Z n) -> fn(Z k) -> sub(chs(n))(k))(25)(42)

Z => (Z => Z) add := fn(Z n) -> fn(Z k) -> sub(chs(n))(k);
add(25)(42)
```

Make sure you have a thorough understanding

- Is the output what you expected?
- Do you see the value of defining variables of type function
- What happens if you omit *sys.reset();* and run again? Try it!
- What happens if you omit the definition of chs?

Funciones, Parámetros, y Reglas de Alcance

Resumen: carguen este código en el editor Zilly y ejecútelo

```
sys.reset();

fn(Z n) -> sub(n)(0)
(fn(Z n) -> sub(n)(0))(18)

Z => Z chs := fn(Z n) -> sub(n)(0);
chs(18)

fn(Z n) -> fn(Z k) -> sub(chs(n))(k)
(fn(Z n) -> fn(Z k) -> sub(chs(n))(k))(25)(42)

Z => (Z => Z) add := fn(Z n) -> fn(Z k) -> sub(chs(n))(k);
add(25)(42)
```

Asegúrense de comprenderlo a fondo.

- ¿Es el resultado el esperado?
- ¿Ven el valor de definir variables de tipo función?
- ¿Qué ocurre si omiten *sys-reset()* y ejecutan el código otra vez?
- ¿Qué ocurre si omiten la definición de chs?

Functions, Parameters, Scope Rules

- **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**
- Note that the nested lambda refers to variable ***n***, as the argument to *chs*, but … ***n*** is nowhere to be found in the nested lambda itself!
- a variable used in an expression (e.g. *n* in *chs(n)*) must be bound to:
  - the closest definition of the variable as a parameter of an enclosing lambda (as is the case here) …
  - … or as a top-level variable, i.e. the ones you define directly, with an initializer
- All variables used in an expression must have been defined somewhere

- The scope of a parameter *p* is the lambda expression where *p* is defined

  … excluding nested scopes that define another parameter with the *same name* *p*

  scope of n: **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**

  scope of k: **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**

- This being an introductory course … we are going to avoid complex topics related to capture of free variables, etc.
- We can always avoid pathological situations with exclusions, because … Barendregt rules apply - we can always rename our variables to avoid conflicts!

Funciones, Parámetros, y Reglas de Alcance

- **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**
- Noten que la lambda anidada hace referencia a la variable ***n***, como argumento de chs, …
  … pero ¡***n*** no se encuentra en ninguna parte de la lambda anidada!
- una variable utilizada en una expresión (p. ej., *n* in *chs(n))* debe estar ligada a:
  - la definición más cercana de la variable, como parámetro de una lambda que la envuelve, …
  - … o como una variable a nivel global, las que Uds. definen directamente, con un inicializador
- Todas las variables utilizadas en una expresión deben estar definidas en algún lugar.

- El alcance de un parámetro *p* es la expresión lambda donde *p* es definida
  … excluyendo ámbitos anidados que definen otro parámetro con el mismo nombre p.
  alcance de **n**: **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**
  alcance de **k**: **fn(Z n) -> fn(Z k) -> sub(chs(n))(k)**

- Por ser este un curso introductorio,
  evitaremos temas complejos relacionados con la captura de variables libres, etc.
- Siempre podemos evitar situaciones patológicas relacionadas con exclusiones, porque …
  (reglas de Barendregt) siempre podemos renombrar nuestras variables y evitar conflictos.

Type Declarations

	Declaration	Description
1	`Z`	Integer
2	`Z => Z`	Function that takes an integer and returns an integer (i.e., a Z to Z)
3	`Z => (Z => Z)`	Function that takes an integer and returns a Z to Z (i.e., a function from a Z to a function from Z to Z)
4	`(Z => Z) => Z`	Function that takes a Z to Z and returns an integer (i.e., a function from a function from Z to Z to a Z)
5	`Z => (Z => (Z => Z))`	Function that takes a Z and returns a {3} (i.e., returns a function that takes a Z and returns a Z => Z)
6	`Z => ((Z => Z) => Z)`	Function that takes a Z and returns a {4} (i.e., returns a function that takes a Z => Z and returns a Z)
7	`(Z => (Z => Z)) => Z`	Function that takes a {3} and returns a Z
8	`((Z => Z) => Z) => Z`	Function that takes a {4} and returns a Z
9	`(Z => Z) => (Z => Z)`	Function that takes a Z => Z and returns a Z => Z (i.e., function that takes a {2} and returns a {2})
…	…	…

Important: the `=>` operator associates to the right.
Therefore `Z => (Z => Z)` can be written as `Z => Z => Z`

Even though we have only one base type (a.k.a. ground type) Z,
we have an infinite (countably infinite to be precise) number of function types.

But we are not going to create artificial computation models to pursue complexity!
The goal of this course is for you to reason with rigor about your computations.

Declaraciones de Tipos

	Declaración	Descripción
1	`Z`	Entero
2	`Z => Z`	Función que toma un entero y retorna un entero (i.e., de Z a Z)
3	`Z => (Z => Z)`	Función que toma un entero y retorna una función de Z a Z (i.e., a función de Z a {2})
4	`(Z => Z) => Z`	Función que toma una función de Z a Z y retorna un entero (i.e., a función de {2} a Z)
5	`Z => (Z => (Z => Z))`	Función que toma Z y retorna {3} (i.e., retorna una función que toma Z y retorna Z => Z)
6	`Z => ((Z => Z) => Z)`	Función que toma Z y retorna {4} (i.e., retorna una función que toma Z => Z y retorna Z)
7	`(Z => (Z => Z)) => Z`	Función que toma {3} y retorna Z
8	`((Z => Z) => Z) => Z`	Función que toma {4} y retorna Z
9	`(Z => Z) => (Z => Z)`	Función que toma Z => Z y retorna Z => Z (i.e., función que toma {2} y retorna {2})
…	…	…

Importante: el operador `=>` asocia a la derecha.
Por lo tanto `Z => (Z => Z)` también puede representarse de esta manera `Z => Z => Z`

A pesar de tener solo un tipo base (tipo "atómico") Z,
hay un número infinito (infinito contable para ser precisos) de tipos de funciones.

Sin embargo no vamos a crear modelos computacionales artificiales para mostrar ejemplos complicados.
El objetivo del curso es que aprendan a razonar de manera rigurosa los cómputos que definen.

Type Inference – Computing Types

- Technically we cheated a few slides ago:
  We did not prove that ***sub(chs(n))(k)*** is of type Z in ***fn(Z n) -> fn(Z k) -> sub(chs(n))(k)***
- Here is the reasoning:
  - ***n*** is of type Z (parameter definition) and ***k*** is of type Z (parameter definition)
  - ***chs*** is of type Z => Z (prove it - hint: *<?> chs := fn(Z n) -> sub(n)(0)*; *sub* has type Z => (Z => Z), so …)
  - ***chs(n)*** is of type Z (because ***n*** is of type Z and ***chs*** is of type Z => Z)
  - ***sub***(chs(n))(k) is of type Z (*sub* arguments are of type Z; sub has type Z => (Z => Z), so …)

Exercises: infer the type of the variables defined here:

- <?> v1 := random(80); (hint: look at the definition of random)
- <?> v2 := fn(Z x) -> sub(1)(v1); (hint: look at the definition of sub)
- <?> v3 := fn(Z x) -> sub(1); (hint: sub has type Z => (Z => Z), therefore sub(Z) returns a …)

Inferencia de Tipos – Computando Tipos

- Técnicamente, hicimos trampa hace unas diapositivas:
  No demostramos que ***sub(chs(n))(k)*** es de tipo Z en ***fn(Z n) -> fn(Z k) -> sub(chs(n))(k)***
- Este es el razonamiento:
  - ***n es de tipo Z (definición del parámetro) y k es de tipo Z (definición del parámetro)***
  - chs es de tipo Z => Z (pista: <?> chs := fn(Z n) -> sub(n)(0); sub tiene tipo Z => (Z => Z), entonces …)
  - chs(n) es de tipo Z (porque n es de tipo Z y chs es de tipo Z => Z)
  - ***sub***(chs(n))(k) tiene tipo Z (ambos argumentos son de tipo Z; sub tiene tipo Z => (Z => Z), entonces …)

Ejercicios: Inferir el tipo de las variables definidas aquí:

- <?> v1 := *random*(80); (pista: vean la definición de *random*)
- <?> v2 := fn(Z x) -> *sub*(1)(v1); (pista: vean la definición de *sub*)
- <?> v3 := fn(Z x) -> *sub*(1); (pista: *sub* tiene tipo Z => (Z => Z), por lo tanto, *sub*(Z) devuelve …)

Type Inference

Exercise: infer the type of ***weird***:

- <?> ***weird*** := fn(Z => Z transformer) -> transformer(random(1000));

First, compute the type of the initializer

- fn(Z => Z transformer) -> transformer(random(1000))

The lambda expression must have a type signature of the form P => R

- where P is the <parameter type> and R is the <result type>

The type of the parameter *transformer* is provided in its definition (Z => Z), so …

- (Z => Z) => transformer(random(1000))

Substitute expressions, bottom up, with their type to derive the solution

- (Z => Z) => transformer(random(Z)) (1000 is an integer)
- (Z => Z) => transformer(Z) (random applied to an integer gives an integer as a result)
- (Z => Z) => Z (transformer has type Z => Z, therefore transformer(Z) has type Z)

Inferencia de Tipos

Ejercicio: Inferir el tipo de ***weird***:

- <?> ***weird*** := fn(Z => Z transformer) -> transformer(random(1000));

Primero, calcula el tipo del inicializador:

- fn(Z => Z transformer) -> transformer(random(1000))

La expresión lambda debe tener una firma de tipo de la forma P => R

- donde P es el <tipo del parámetro> y R es el <tipo del resultado>

El tipo del parámetro *transformer* se proporciona en su definición (Z => Z), así que …

- (Z => Z) => transformer(random(1000))

Sustituir las expresiones, de abajo a arriba, por su tipo, hasta obtener la solución

- (Z => Z) => transformer(random(Z)) (1000 es un entero)
- (Z => Z) => transformer(Z) (random aplicado a un entero da como resultado un entero)
- (Z => Z) => Z (transformer tiene tipo Z => Z, por lo tanto, transformer(Z) tiene tipo Z)

Type Inference

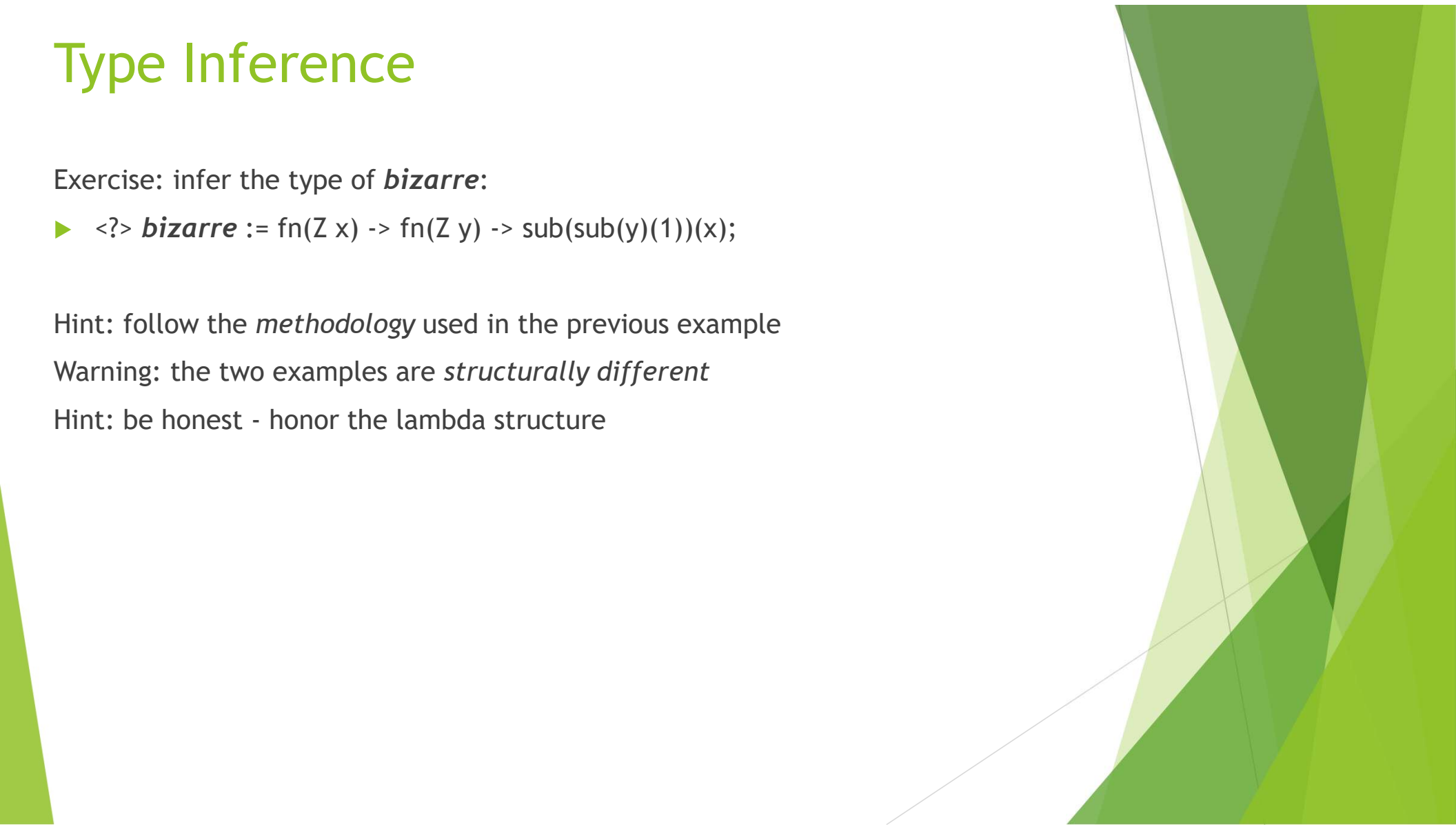
Exercise: infer the type of ***bizarre***:

- <?> ***bizarre*** := fn(Z x) -> fn(Z y) -> sub(sub(y)(1))(x);

Hint: follow the *methodology* used in the previous example

Warning: the two examples are *structurally different*

Hint: be honest - honor the lambda structure



Inferencia de Tipos

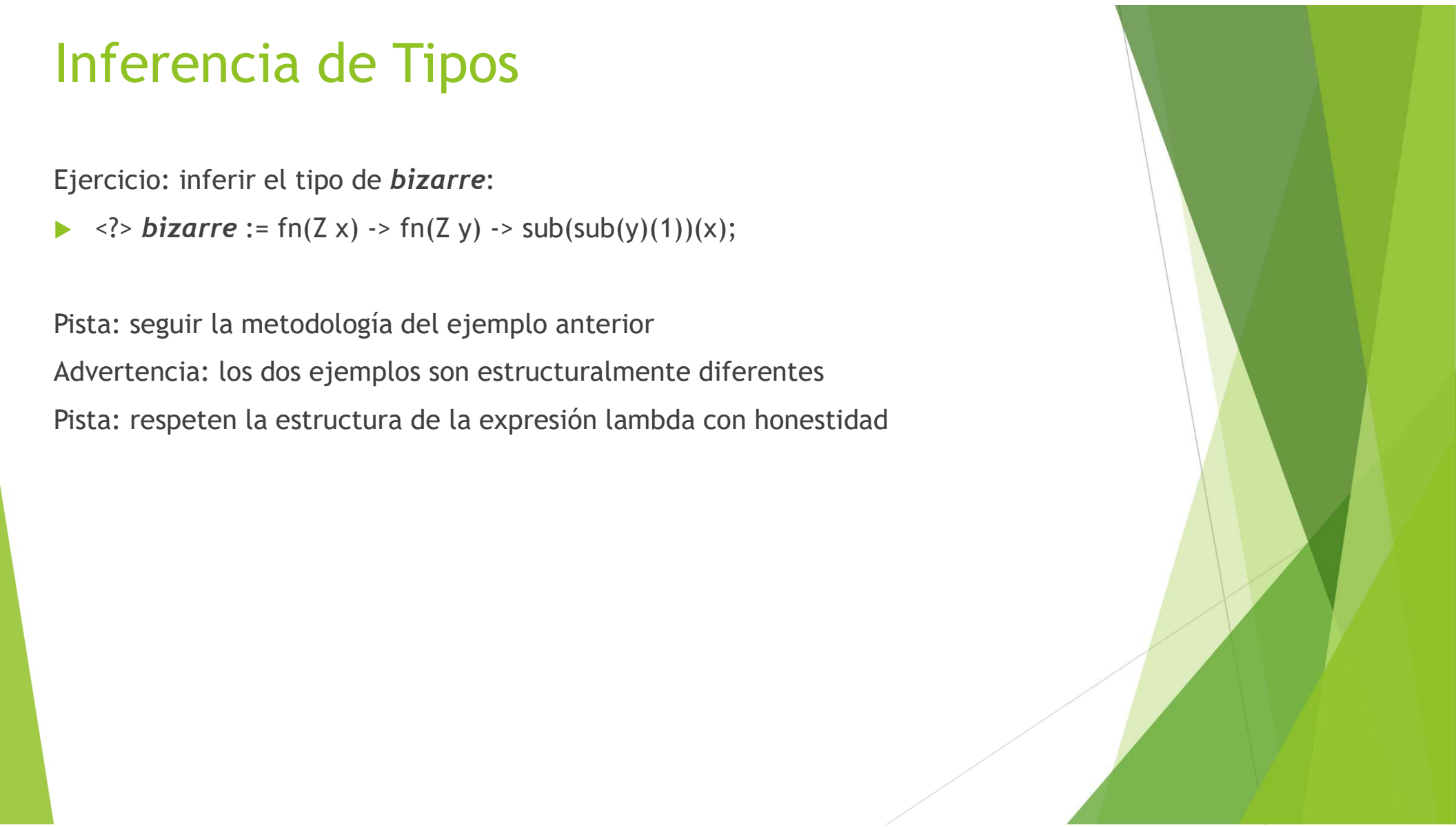
Ejercicio: inferir el tipo de ***bizarre***:

- <?> ***bizarre*** := fn(Z x) -> fn(Z y) -> sub(sub(y)(1))(x);

Pista: seguir la metodología del ejemplo anterior

Advertencia: los dos ejemplos son estructuralmente diferentes

Pista: respeten la estructura de la expresión lambda con honestidad



Partial Application

Step 1. Don't run … read the code below, making an honest effort to understand it, and explain what it does
Step 2. Copy the code, paste it on the Zilly Editor, and run
Step 3. Check the results, comparing them to your conclusions on step 1

```
sys.reset();

Z => (Z => Z) lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);

lpf(120)(109)
lpf(120)(119)
lpf(120)(120)
lpf(120)(130)
```

Aplicación Parcial

Paso 1. No se apuren ... lean el código, haciendo un buen esfuerzo para comprenderlo y explicar qué hace
Paso 2. Copien el código al editor de Zilly y ejecútenlo
Paso 3. Comprueben los resultados, compárenlos con sus conclusiones en el paso 1

```
sys.reset();

Z => (Z => Z) lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);

lpf(120)(109)
lpf(120)(119)
lpf(120)(120)
lpf(120)(130)
```

Partial Application

*Now, note the **addition** of function **lpf120**, and pay particular attention to its **initialization**.*
***This** is an example of **partial application**, the previous one wasn't. **Partial application** is a useful concept*

```
sys.reset();

Z => Z => Z lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);

lpf(120)(109)
lpf(120)(119)
lpf(120)(120)
lpf(120)(130)

Z => Z lpf120 := lpf(120);

lpf120(109)
lpf120(119)
lpf120(120)
lpf120(130)
```

Aplicación Parcial

Ahora, noten la adición de la función lpf120 y preste especial atención a su inicialización.
***Este es un ejemplo de aplicación parcial**, el anterior no lo era. La **aplicación parcial** es un concepto útil.*

```
sys.reset();

Z => Z => Z lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);

lpf(120)(109)
lpf(120)(119)
lpf(120)(120)
lpf(120)(130)

Z => Z lpf120 := lpf(120);

lpf120(109)
lpf120(119)
lpf120(120)
lpf120(130)
```

Partial Application

Note these two variable definitions. Do they compute the same function?
Pay particular attention to the parameters and their use in the body
Remember: renaming function parameters (*) does NOT change a function semantics (**)
(*) this assumes to you are updating their use in the body accordingly, of course!
(**) in lambda calculus, the renaming of parameters is known as alpha-reduction

- *Z => Z => Z lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);*
- *Z => Z => Z min := fn(Z y) -> fn(Z x) -> if(lt(y)(x), x, y);*

Aplicación Parcial

Consideren estas dos definiciones de variables. ¿Calculan la misma función?
Presten atención especial a los parámetros y su uso en el cuerpo de la función.
Recuerden: renombrar los parámetros de una función (*) NO cambia la semántica de la función (**).

(*) Esto presupone que está actualizando su uso correspondientemente en el cuerpo, ¡por supuesto!
(**) En cálculo lambda, renombrar los parámetros se conoce como alfa-reducción.

- *Z => Z => Z lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);*
- *Z => Z => Z min := fn(Z y) -> fn(Z x) -> if(lt(y)(x), x, y);*

Partial Application

Look at the code here and compare it to the partial application example.
Do we still need lpf to define lpf120? Why or why not?

```
sys.reset();

Z => Z => Z lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);

lpf(120)(109)
lpf(120)(119)
lpf(120)(120)
lpf(120)(130)

Z => Z => Z min := fn(Z y) -> fn(Z x) -> if(lt(y)(x), x, y);
Z => Z lpf120 := min(120);

lpf120(109)
lpf120(119)
lpf120(120)
lpf120(130)
```

Aplicación Parcial

Observen este código y compárenlo con el ejemplo de aplicación parcial.
¿Aún necesitamos lpf para definir lpf120? ¿Por qué sí o por qué no?

```
sys.reset();

Z => Z => Z lpf := fn(Z max) -> fn(Z volts) -> if(lt(max)(volts), volts, max);

lpf(120)(109)
lpf(120)(119)
lpf(120)(120)
lpf(120)(130)

Z => Z => Z min := fn(Z y) -> fn(Z x) -> if(lt(y)(x), x, y);
Z => Z lpf120 := min(120);

lpf120(109)
lpf120(119)
lpf120(120)
lpf120(130)
```

Partial Application Exercises

- Consider the following *hypothetical* definition of the primitive function ***sub:***
  - Z => Z => Z ***sub*** := fn(Z x) -> fn(Z y) -> ***y - x***;
- Why is the definition *hypothetical*? Because ***sub*** is a *primitive* function in the current implementation of Zilly. Two functions, ***sub*** and ***lt***, are the basic ingredients (the ***basis*** of primitive functions) that generate the entire system of integer arithmetic. We can't define ***sub*** from user-derived functions, but we can describe ***sub*** as we did above. Note the type declaration: ***sub*** and ***lt*** have the same type signature of the user-derived functions ***add*** and ***min***: Z => Z => Z. Also, the body is shown in red because it describes the semantics of ***sub*** using mathematics as the denotation language … it's like implementing ***sub*** in math. ☺
- From the *function application* perspective, ***sub*** is no different from other functions with the same type signature: for instance, it can be partially evaluated.

1. Exercises:(Elegant Solution) Define the predecessor function ***pred***, from ***sub***, using partial evaluation
2. Imagine replacing ***sub*** with a function named ***minus***, forming a Zilly VM with a new basis: [lt, minus]

   Function minus is described here: Z => Z => Z ***minus*** := fn(x) -> fn(y) -> ***x - y***;

   a) If ***minus*** is in the basis, can you derive ***sub*** from it?

   b) If ***minus*** is in the basis, can you implement your elegant solution in (1) using only ***minus*** directly?

   c) If ***sub*** is in the basis, can you derive ***minus*** from it?

   d) Do the Zilly [lt, sub] VM and the Zilly [lt, minus] VM have the same power?

Ejercicios de Aplicación Parcial

- Consideren esta definición hipotética de la función primitiva ***sub:***
  - Z => Z => Z ***sub*** := fn(Z x) -> fn(Z y) -> ***y - x***;
- ¿Por qué es esta definición *hipotética*? Porque ***sub*** es una función primitiva en la implementación de Zilly. Dos funciones, ***sub*** y ***lt***, conforman la ***base*** funcional para generar todo el sistema de aritmética de enteros. No podemos definir ***sub*** a partir de funciones derivadas del usuario, pero podemos describirla como lo hacemos aquí. Noten la declaración de tipo: ***sub*** and ***lt*** tienen la misma signatura de tipo que las funciones derivadas ***add*** y ***min***: Z => Z => Z. Además, el cuerpo se muestra en rojo porque la semántica de ***sub*** se describe usando el lenguaje de las matemáticas como lenguaje de denotación … es como implementar ***sub*** en matemáticas.
- Desde la perspectiva de aplicación de funciones, ***sub*** no es distinta de otras funciones con la misma signatura de tipo: por ejemplo, se puede evaluar parcialmente.

1. (Solución elegante) Defina la función predecesora ***pred***, a partir de sub, mediante evaluación parcial.
2. Imagine reemplazar ***sub*** con una función llamada ***minus***, formando una nueva ***base*** [***lt***, ***minus***] para Zilly

   La función ***minus*** se describe: Z => Z => Z ***minus*** := fn(Z x) -> fn(Z y) -> ***x - y***; (noten que ***no*** es igual a sub)

   a) Si ***minus*** es parte de la base, ¿pueden derivar ***sub*** a partir de ella?

   b) Si ***minus*** es parte de la base, ¿pueden implementar la solución elegante en (1) utilizando solo ***minus*** directamente?

   c) Si ***sub*** es parte de la base, ¿pueden derivar ***minus*** a partir de ella?

   d) ¿Tienen las máquinas virtuales Zilly [*lt*, *sub*] y Zilly [***lt***, ***minus***] la misma potencia?

Recursion

- Run the code below in the Zilly ICE, as usual.
- And make an honest effort to understand the code.
- What does sum1stN compute? Note the use of recursion.

```
sys.reset();

Z => Z chs := fn(Z n) -> sub(n)(0);
Z => Z => Z add := fn(Z n) -> fn(Z k) -> sub(chs(k))(n);
Z => Z sum1stN := fn(Z n) -> if(lt(0)(n), 0, add(sum1stN(sub(1)(n)))(n));

sum1stN(0)
sum1stN(1)
sum1stN(2)
sum1stN(5)
sum1stN(10)
sum1stN(100)
sum1stN(1000)
```

Recursión

- Ejecute el código a continuación en el Zilly ICE, como de costumbre.
- Esfuércese por comprender el código.
- ¿Qué calcula sum1stN? Observe el uso de recursión en el código.

```
sys.reset();

Z => Z chs := fn(Z n) -> sub(n)(0);
Z => Z => Z add := fn(Z n) -> fn(Z k) -> sub(chs(k))(n);
Z => Z sum1stN := fn(Z n) -> if(lt(0)(n), 0, add(sum1stN(sub(1)(n)))(n));

sum1stN(0)
sum1stN(1)
sum1stN(2)
sum1stN(5)
sum1stN(10)
sum1stN(100)
sum1stN(1000)
```

A little **sugar**, please … we want **Shugar!!!**

```
::zilly+
sys.reset();

Z x := 6;
Z y := 7;
x < y
x <= y
x = y
x <> y
x >= y
x > y

x + 1
y - 1
x * y
```

- The code on the left demonstrates the Zilly+ VM extension
- To enable it, you just add the command **::zilly+** at the top.
- It provides operators, using binary infix notation, for:
  - number comparison
  - addition, subtraction, and multiplication
- To enjoy the new sugar, evaluate the following expressions:
  - (x + 1) * (y – 1)
  - y < (x + 2)
- And it comes at not extra charge for students! ☺

Algo de **azúcar**, por favor … queremos **Azúkar!!!**

```
::zilly+
sys.reset();

Z x := 6;
Z y := 7;
x < y
x <= y
x = y
x <> y
x >= y
x > y

x + 1
y - 1
x * y
```

- El código a la izquierda muestra el uso de la extensión Zilly+
- Para habilitarla, agregue el comando **::zilly+** al comienzo.
- Provee operadores, usando notación binaria infija, para:
  - comparación de números
  - suma, resta y multiplicación
- Para disfrutar el nuevo azúcar, evalúe estas expresiones:
  - (x + 1) * (y – 1)
  - y < (x + 2)
- ¡Y todo esto es gratis para los estudiantes! ☺

Recursion

- sum1stN computes the sum of the first N positive integers
- Do you find easier to understand the code in the conventional sugared syntax?

```
::zilly+
sys.reset();

Z => Z sum1stN := fn(Z n) -> if(n < 0, 0, n + sum1stN(n - 1));

sum1stN(0)
sum1stN(1)
sum1stN(2)
sum1stN(5)
sum1stN(10)
sum1stN(100)
sum1stN(1000)
```

Recursión

- sum1stN calcula la suma de los primeros N números enteros positivos
- ¿Le resulta más fácil comprender el código con la sintaxis convencional?

```
::zilly+
sys.reset();

Z => Z sum1stN := fn(Z n) -> if(n < 0, 0, n + sum1stN(n - 1));

sum1stN(0)
sum1stN(1)
sum1stN(2)
sum1stN(5)
sum1stN(10)
sum1stN(100)
sum1stN(1000)
```