Computing with Zilly

A practical introduction to the Zilly language

Enzo Alda & Daniel Andrés Pinto Alvarado

Computando con Zilly

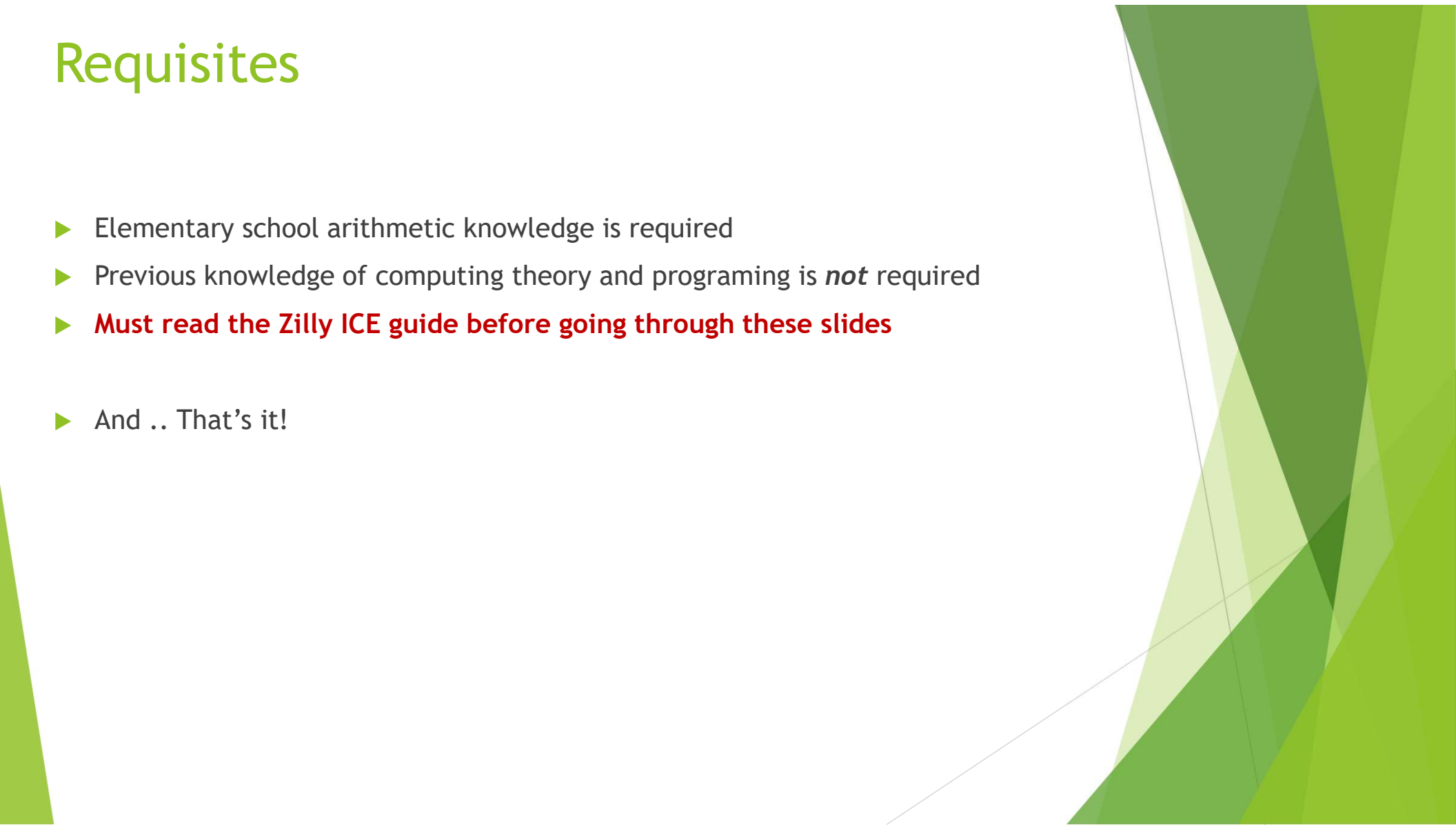
Una introducción práctica al lenguaje Zilly

Enzo Alda & Daniel Andrés Pinto Alvarado

Requisites

- Elementary school arithmetic knowledge is required
- Previous knowledge of computing theory and programing is ***not*** required
- **Must read the Zilly ICE guide before going through these slides**

- And .. That's it!



Requisitos

- Se requieren conocimientos de aritmética de primaria.
- No se requieren conocimientos previos de teoría informática ni programación.
- **Es imprescindible leer la guía Zilly ICE antes de ver estas láminas.**

- Y .. ¡Eso es todo!

Part I

Parte I

Evaluating expressions

- As you already know, Zilly computes with integer numbers
- There are two important functions we can use in expressions: ***lt*** and ***sub***
- Their semantics (i.e., their meaning, what they compute) is as follows:

  lt(n)(k) ➔ ***k < n*** (specifically, 1 if *k < n* and 0 otherwise)

  sub(n)(k) ➔ ***k - n*** (i.e., subtract n from k)
- Turns out, those two operations are all you need to implement integer arithmetic!

- There are a few other predefined functions: ***random*** is useful for testing:

  random*(n)* ➔ (0 if *n* = 0, otherwise, a random integer between *0* and *n-1 included*)

- Some predefined functions are system functions, prefixed by *sys*, like sys.reset

  Executing sys.reset() erases all user-defined variables.

Evaluando expresiones

- Como ya saben, Zilly realiza cálculos con números enteros.
- Hay dos funciones importantes que podemos usar en expresiones: ***lt*** y ***sub***
- Su semántica (es decir, su significado, lo que calculan) es la siguiente:

  lt(n)(k) ➔ ***k < n*** (específicamente, 1 si *k < n* y 0 en caso contrario).

  sub(n)(k) ➔ ***k - n*** (es decir, resta n de k).
- Esas dos operaciones son todo lo que necesitan para implementar la aritmética de enteros

- Hay otras funciones predefinidas: ***random*** es útil para realizar pruebas:

  random(n) ➔ (0 si n = 0; de lo contrario, un entero aleatorio entre 0 y n-1 incluido)

- Algunas funciones predefinidas son funciones de sistema, prefijadas con sys, como sys.reset

  Al ejecutar sys.reset() se borran todas las variables definidas por el usuario

Sample programs: evaluating expressions

Copy and paste the sample code on the left into the Zilly editor window, then hit the Run button
Always read the output and strive to understand it: e.g., **Q: sub(7)(4)** (Query: subtract 7 from 4)
R: OK: sub(7)(4) ==> -3 (Reply: OK, subtracting 7 from 4 yields -3 as the result)
Repeat for the code on the right. From now on, always do the same when encountering sample code.

```
0
1
sub(4)(7)
sub(7)(4)
lt(4)(4)
lt(4)(7)
lt(7)(4)
lt(7)(7)
```

```
random(8)
random(8)
random(8)
random(8)
random(8)
random(8)
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
```

Programas de ejemplo: evaluando expresiones

Copia y pega el código de ejemplo a la izquierda en la ventana del editor Zilly y pulsa el botón “Run”.
Siempre lee el resultado y esfuérzate por comprenderlo: p. ej., **Q: sub(7)(4)** (Consulta: restar 7 de 4)
R: OK: sub(7)(4) ==> -3 (Respuesta: OK, restar 7 de 4 da como resultado -3)
Repite el proceso para el código a la derecha. De ahora en adelante, haz lo mismo cuando encuentres código de ejemplo.

```
0
1
sub(4)(7)
sub(7)(4)
lt(4)(4)
lt(4)(7)
lt(7)(4)
lt(7)(7)
```

```
random(8)
random(8)
random(8)
random(8)
random(8)
random(8)
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
lt(4)(random(8))
```

How to define an integer variable in Zilly

- The following command defines an **integer** variable named ***k*** with value ***8*:**

  Z k := 8;

- The command can be read as "define an **integer** variable **k** initializing it with **8**"
  - **Z** represents the *integer type*
  - **k** is the *name* (a.k.a. symbol) of the variable
  - := is known as the *assignment operator* – indicates that we are assigning a value to the variable
  - 8 is the expression used to initialize the variable, known as the "*initializer*"
  - ; is the statement termination symbol, also known as the "*terminator*"

- We call this kind of command a "***variable definition***"
- The general syntax of a variable definition in Zilly is:

  <*type*> <*symbol*> := <*expression*>;

- Note that the initializer is an *expression* in general

  it can be an integer value, like *8*

  it can be a complex expression, like *sub(7)(random(100))*

Cómo definir una variable entera en Zilly

- El siguiente comando define una variable entera llamada k con valor ***8*:**

  Z k := 8;

- El comando puede leerse como “define una variable entera ***k*** inicializándola con 8”
  - **Z** representa el *tipo entero*
  - **k** es el *nombre* (símbolo) de la variable
  - := es el *operador de asignación* – indica que estamos asignando un valor a la variable
  - 8 es la expresión utilizada para inicializar la variable, conocida como “*inicializador*”
  - ; es el símbolo de terminación de la instrucción, también conocido como “*terminador*”

- A este tipo de comando lo llamamos “***definición de variable***”
- La sintaxis general de une definición de variable en Zilly es:

  <*tipo*> <*símbolo*> := <*expresión*>;

- Tenga en cuenta que el inicializador es una *expresión* en general

  Puede ser un valor entero, como *8*

  Puede ser una expresión compleja, como *sub(7)(random(100))*

Defining integer variables: a guided exercise in 2 steps

1. Do the following, using the command line in the Zilly Web REPL
   - First, click the reset button `00` to reset the Zilly VM memory, so you can start from scratch
   - Another (brute force) way to start from scratch is refreshing your browser using the F5 key
   - Enter and submit the variable definition we used as an example in the previous slide:

     Z k := 8;
   - Evaluate *k* (just enter *k* in the command line and submit: a variable is a valid expression)

2. Now (do not reset the VM - if you did so, k is no longer defined: you will have to redo part 1) …
   - Execute this variable definition:

     Z x := sub(3)(k);
   - Evaluate x
   - Evaluate k
   - Evaluate sub(x)(k)
   - Evaluate sub(k)(x)

- Always read the output and make sure you ***understand*** what's happening.

Defining integer variables: a guided exercise in 2 steps

1. Haz lo siguiente usando la línea de comandos en el REPL web de Zilly
   - Primero, haz clic en el botón `00` para reiniciar la memoria de la máquina virtual desde cero
   - Otra forma de empezar desde cero (a fuerza bruta) es actualizar el navegador con la tecla F5
   - Introduce y envía la definición de variable que usamos como ejemplo en la diapositiva anterior:

     Z k := 8;
   - Evalúa k (simplemente introduce k en la línea de comandos y envíala: una variable es una expresión válida)

2. Ahora (sin reiniciar la máquina virtual, de otra forma tendrás que rehacer el paso 1) …
   - Ejecuta esta definición de variable:

     Z x := sub(3)(k);
   - Evalúa x
   - Evalúa k
   - Evalúa sub(x)(k)
   - Evalúa sub(k)(x)

- Siempre lee la salida y asegúrate de comprender lo que esta pasando

Defining integer variables

You already know what to do with sample code: read and understand!
You can also use the command line to evaluate variables, and expressions that use them.
Experimenting with the ICE will help you learn more and faster.
Ask yourself questions, like: “can I infer the value of r8y from the result of sub(r8x)(r8y)?”
Note the use of *sys.reset()* at the top, to reset the VM on every run.

```
sys.reset();
Z xcd := 42;
Z yss := 67;
Z ryx := sub(yss)(xcd);
Z rxy := sub(xcd)(yss);
Z xng := sub(xcd)(sub(xcd)(xcd));
```

```
sys.reset();
Z r8x := random(8);
r8x
r8x
r8x
Z r8y := random(8);
sub(r8x)(r8y)
sub(r8x)(r8y)
sub(r8x)(r8y)
sub(r8x)(r8y)
```

Definiendo variables enteras

Ya sabes qué hacer con código de ejemplo: ¡leerlo y comprenderlo!
También puedes usar la línea de comandos para evaluar variables y expresiones.
Experimentar con el ICE te ayudará a aprender más y más rápido.
Hazte preguntas como: "¿Puedo inferir el valor de r8y a partir del resultado de sub(r8x)(r8y)?".
Observa el uso de sys.reset() al comienzo, para reiniciar la máquina virtual en cada ejecución.

```
sys.reset();
Z xcd := 42;
Z yss := 67;
Z ryx := sub(yss)(xcd);
Z rxy := sub(xcd)(yss);
Z xng := sub(xcd)(sub(xcd)(xcd));
```

```
sys.reset();
Z r8x := random(8);
r8x
r8x
r8x
Z r8y := random(8);
sub(r8x)(r8y)
sub(r8x)(r8y)
sub(r8x)(r8y)
sub(r8x)(r8y)
```

Appetizer: a digression about function types

- Given that Zilly only computes with integer numbers you may be wondering:

  "why is it necessary to specify the type of the variables? Aren't all of them integers?"
- Turns out variables can represent not only numbers but functions as well!
- In fact, ***sub***, ***lt***, and ***random*** are *predefined* (internal) *variables* of a ***function type***
- The ***random*** function takes an integer as argument and returns an integer as the result
  - Therefore, *random* has type *Z => Z*, because it maps an integer to an integer
  - *Z => Z random := <low level internal code>;*
- Functions ***lt*** and ***sub*** are of type *Z => (Z => Z)*, which is the same as *Z => Z => Z*
  - *Z => Z => Z lt := <low level internal code>;*
  - *Z => Z => Z sub := <low level internal code>;*
- There are *infinite* (countably infinite) function types … but we only use a few!
- Functions can take functions as arguments and return functions as result

  For instance, if a function ***f*** has type ***(Z => Z) => Z => (Z => Z)*** it means that …
  … given a function of type ***(Z => Z)*** and an integer, ***f*** returns a function of type ***(Z => Z)***

  Don't worry if this is not entirely clear to you: you will learn more about this topic later …
  … in the theory lectures (Frege, Schönfinkel, Curry) and exploring advanced functional abstraction

Aperitivo: una digresión sobre los tipos de funciones

- Dado que Zilly sólo calcula con números enteros, puede que te preguntes:

  "¿Por qué es necesario especificar el tipo de las variables? ¿No son todas enteras?".
- Resulta que las variables pueden representar no solo números, sino también funciones.
- De hecho, ***sub***, ***lt*** y ***random*** son variables predefinidas (internas) de un ***tipo función***
- La función ***random*** toma un entero como argumento y devuelve un entero como resultado.
  - Por lo tanto, *random* tiene tipo *Z => Z*, ya que computa un entero a partir de otro entero
  - *Z => Z random := <código interno de bajo nivel>;*
- Las funciones ***lt*** y ***sub*** son de tipo *Z => (Z => Z)*, que es lo mismo que *Z => Z => Z*
  - *Z => Z => Z lt := <código interno de bajo nivel>;*
  - *Z => Z => Z sub := <código interno de bajo nivel>;*
- Existen infinitos tipos de funciones (contablemente infinitos), ¡pero solo usamos unos pocos!
- Las funciones pueden tomar funciones como argumentos y devolver funciones como resultado.

  Por ejemplo, si una función ***f*** tiene tipo ***(Z => Z) => Z => (Z => Z)***, significa que …
  … dada una función de tipo ***(Z => Z)*** y un entero, ***f*** retorna una función de tipo ***(Z => Z)***

  No te preocupes si esto no es del todo claro: aprenderás más sobre este tema adelante …
  … en las clases de teoría (Frege, Schönfinkel, Curry) y explorando la abstracción funcional avanzada.

Conditional Expressions

- As is the case in life, complex computations require making decisions
- For instance, if we need to choose the smaller of two numbers, x and y, then …
  - In C, C++, Java, C#, JavaScript, and many other C-inspired languages we write:
    - (x < y ? x : y)
  - In Python we write:
    - (x if x < y else y)
  - In LISP we write
    - if ((< x y) x y)
  - In MS-Excel, Google Sheets, Gnumeric, and pretty much *any* spreadsheet we write:
    - if(x < y, x, y)
- **All** the syntactic "sugars" (**forms**) above require exactly three expressions:
  - A Boolean expression, known as the *condition*, which is evaluated first
  - An expression (known as *consequent*), evaluated if and only if the *condition* is true
  - An expression (known as *alternative*), evaluated if and only if the *condition* is false
  - (different sugars, same semantics … get used to focus on semantics)

Expresiones Condicionales

- Al igual que en la vida real, los cálculos complejos requieren tomar decisiones
- Por ejemplo, si necesitamos escoger el menor de dos números, x e y, entonces …
  - En C, C++, Java, C#, JavaScript y muchos otros lenguajes inspirados en C, escribimos:
    - (x < y ? x : y)
  - En Python escribimos:
    - (x if x < y else y)
  - En LISP escribimos:
    - if ((< x y) x y)
  - En MS-Excel, Google Sheets, Gnumeric y prácticamente cualquier hoja de cálculo, escribimos:
    - if(x < y, x, y)
- **Todos** los "azúcares" sintácticos (**formas**) anteriores requieren exactamente tres expresiones:
  - Una expresión booleana, conocida como *condición*, que se evalúa primero
  - Una expresión (*consecuente*), que se evalúa si y solo si la *condición* es verdadera
  - Una expresión (*alternativa*), que se evalúa si y solo si la *condición* es falsa
  - (Diferentes expresiones, con la misma semántica … enfóquense en la semántica)

Conditional Expressions

- The syntax of conditional expressions in Zilly mirrors that of ... ***spreadsheets*** [1]:

  if (< condition >, < consequent >, < alternative >)

- Logically, the *condition* is considered a Boolean expression, but …
  … there is no Boolean type in Zilly! How can then we have conditional expressions?
- Simple, we use the values 0 and 1 to represent the Boolean values *false* and *true*
  - The same convention was adopted by C decades ago, and by logicians over a century ago
- The *condition* must be an integer, but there are many integers other than 0 and 1
- The *condition* is ***false*** if it evaluates to 0, ***true*** otherwise (same as in C, for over 5 decades)
- The *consequent* and the *alternative* can have *any type*, but must have the *same type*
- Knowing that the math expression ***x < y*** is written ***lt(y)(x)*** in Zilly we can finally write:

  if(lt(y)(x), x, y)

  Try it now, in the REPL, for different values of x and y values – Even better: try it with your own variables!

---

[1] For historical reasons - turns out, the USB is a pioneer in advanced spreadsheet computing ☺
Towards Wide-Spectrum Spreadsheet Computing | IEEE Xplore

Expresiones Condicionales

- La sintaxis de las expresiones condicionales en Zilly refleja la de las hojas de cálculo [1]:

  if (< condición >, < consecuente >, < alternativa >)

- Lógicamente, la *condición* se considera una expresión booleana, pero …
  … ¡no existe un tipo booleano en Zilly! ¿Cómo podemos entonces tener expresiones condicionales?
- Simplemente, usamos los valores 0 y 1 para representar los valores booleanos *falso* y *verdadero*
  - Esta misma convención fue adoptada por C hace décadas y por lógicos/matemáticos hace más de un siglo
- La *condición* debe ser un entero, pero existen muchos enteros además de 0 y 1
- La *condición* es ***falsa*** si se evalúa como 0, ***verdadera*** en caso contrario (como en C, por 5+ décadas)
- El *consecuente* y la *alternativa* pueden ser de *cualquier tipo*, pero deben tener *el mismo tipo*
- Sabiendo que la expresión matemática $x < y$ se escribe ***lt(y)(x)*** en Zilly, al fin podemos escribir:

  if(lt(y)(x), x, y)

  Pruébala ya, en el REPL, para diferentes valores de x e y. Mejor aún: ¡pruébala con tus propias variables!

---

[1] Por razones históricas, resulta que la USB es pionera en la computación avanzada con hojas de cálculo ☺
Towards Wide-Spectrum Spreadsheet Computing | IEEE Xplore

Conditional expressions: examples to exercise your reasoning!

Consider the program on the **left**:

- What can we say about z when the expression ***if(lt(z)(x), if(lt(z)(y), 1, 0), 0)*** returns true?
- Describe what ***if(lt(y)(x), if(lt(z)(y), z, y), if(lt(z)(x), z, x))*** computes.

In the program on the **right**, interpret the a0, a1, a2 variables as elements of a sequence.

- What can we assert about that sequence when ***if(lt(a1)(a0), if(lt(a2)(a1), 1, 0), 0)*** returns true?

Experimenting in the Zilly ICE and asking questions like the ones above will help you with the quiz.

```
sys.reset();
Z x := random(100);
Z y := random(100);
Z z := random(100);
x
y
z
if(lt(z)(x), if(lt(z)(y), 1, 0), 0)
if(lt(y)(x), if(lt(z)(y), z, y), if(lt(z)(x), z, x))
```

```
sys.reset();
Z a0 := random(100);
Z a1 := random(100);
Z a2 := random(100);
a0
a1
a2
if(lt(a1)(a0), if(lt(a2)(a1), 1, 0), 0)
```

Expresiones condicionales: ¡ejemplos para ejercitar su razonamiento!

Considere el programa del lado **izquierdo**:

- ¿Qué podemos decir sobre z cuando la expresión ***if(lt(z)(x), if(lt(z)(y), 1, 0), 0)*** retorna verdadero?
- Describa qué calcula la expresión ***if(lt(y)(x), if(lt(z)(y), z, y), if(lt(z)(x), z, x))***

En el programa de la **derecha**, interprete las variables a0, a1, a2 como elementos de una secuencia

- ¿Qué podemos afirmar sobre esa secuencia cuando ***if(lt(a1)(a0), if(lt(a2)(a1), 1, 0), 0)*** retorna verdadero?

Experimentar con el Zilly ICE y hacer preguntas como las anteriores le ayudará en el quiz.

```
sys.reset();
Z x := random(100);
Z y := random(100);
Z z := random(100);
x
y
z
if(lt(z)(x), if(lt(z)(y), 1, 0), 0)
if(lt(y)(x), if(lt(z)(y), z, y), if(lt(z)(x), z, x))
```

```
sys.reset();
Z a0 := random(100);
Z a1 := random(100);
Z a2 := random(100);
a0
a1
a2
if(lt(a1)(a0), if(lt(a2)(a1), 1, 0), 0)
```

Functional Abstraction: defining functions
step 0: recognize repetitive patterns and “long sausages”

```
sub(1)(43) ➔ 42
sub(1)(1) ➔ 0
sub(1)(0) ➔ -1
```

```
sub(1)(x) ➔ x - 1
```

- Sometimes we see repetitive patterns - in the case above, subtracting 1 to a number
- Sometimes we have an expression we would like to make shorter and easier to understand
- Functional abstraction means defining functions that abstract computation details
- It is a basic ingredient of the ***art*** and ***science*** of making programs better:
  - Easier to understand
  - Easier to maintain
  - Easier to test

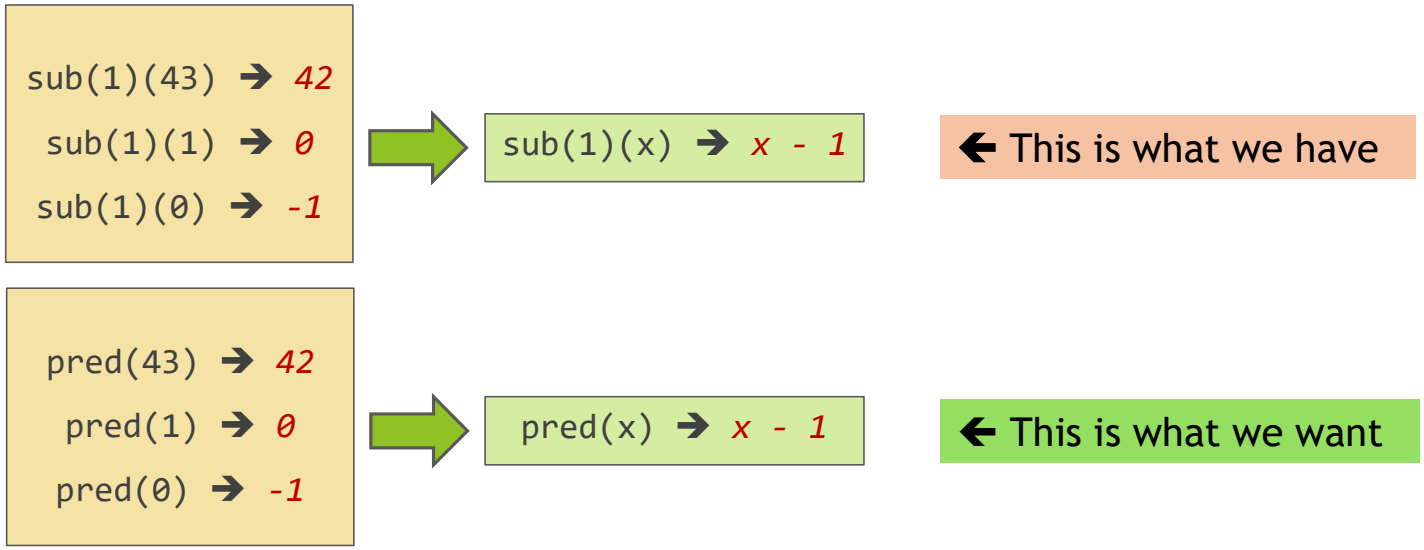
Abstracción funcional: definición de funciones paso 0: reconocer patrones repetitivos y “chorizos largos”

```
sub(1)(43) ➔ 42
sub(1)(1) ➔ 0
sub(1)(0) ➔ -1
```

```
sub(1)(x) ➔ x - 1
```

- A veces vemos patrones repetitivos; en el caso anterior, restar 1 a un número
- A veces tenemos una expresión que nos gustaría acortar para facilitar su comprensión
- La abstracción funcional implica definir funciones que abstraen los detalles del cálculo
- Es un ingrediente básico del ***arte*** y la ***ciencia*** de mejorar los programas:
  - Más fácil de entender
  - Más fácil de mantener
  - Más fácil de probar

Functional Abstraction: defining functions
step 1: identify a desirable and meaningful abstraction



- Wouldn't it be nice to have a function named ***pred***, that computes the predecessor of a number?
  - It only requires one argument
  - The name makes clear its purpose
- Yes, but we can't directly tell *Zilly* "give me a function ***pred***, such that ***pred(x)*** ➔ ***x - 1***" or …
  - … "give me a function ***pred*** that computes the predecessor" … *Zilly* is not AI
- We need to define ***pred*** in terms of something the VM already understands, and that's step 2

Abstracción funcional: definición de funciones
paso 1: identificar una abstracción deseable y significativa

```
sub(1)(43) ➔ 42
sub(1)(1) ➔ 0
sub(1)(0) ➔ -1
```

```
sub(1)(x) ➔ x - 1
```

🡐 Esto es lo que tenemos

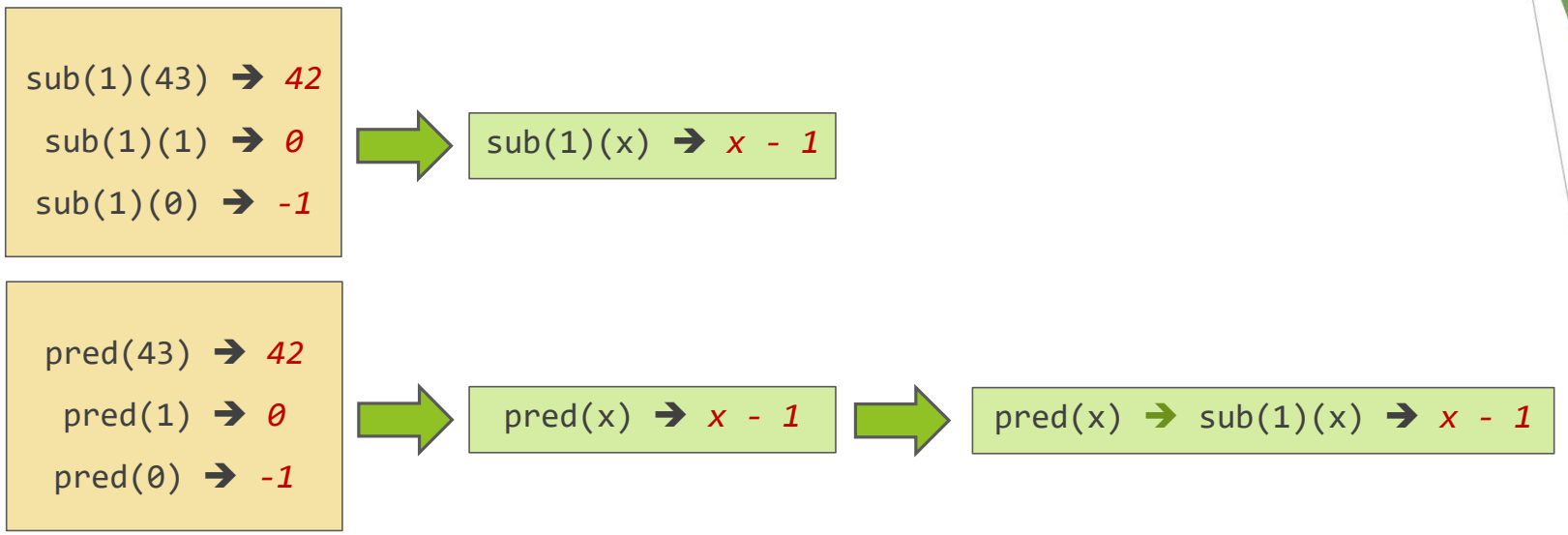
```
pred(43) ➔ 42
pred(1) ➔ 0
pred(0) ➔ -1
```

```
pred(x) ➔ x - 1
```

🡐 Esto es lo que queremos

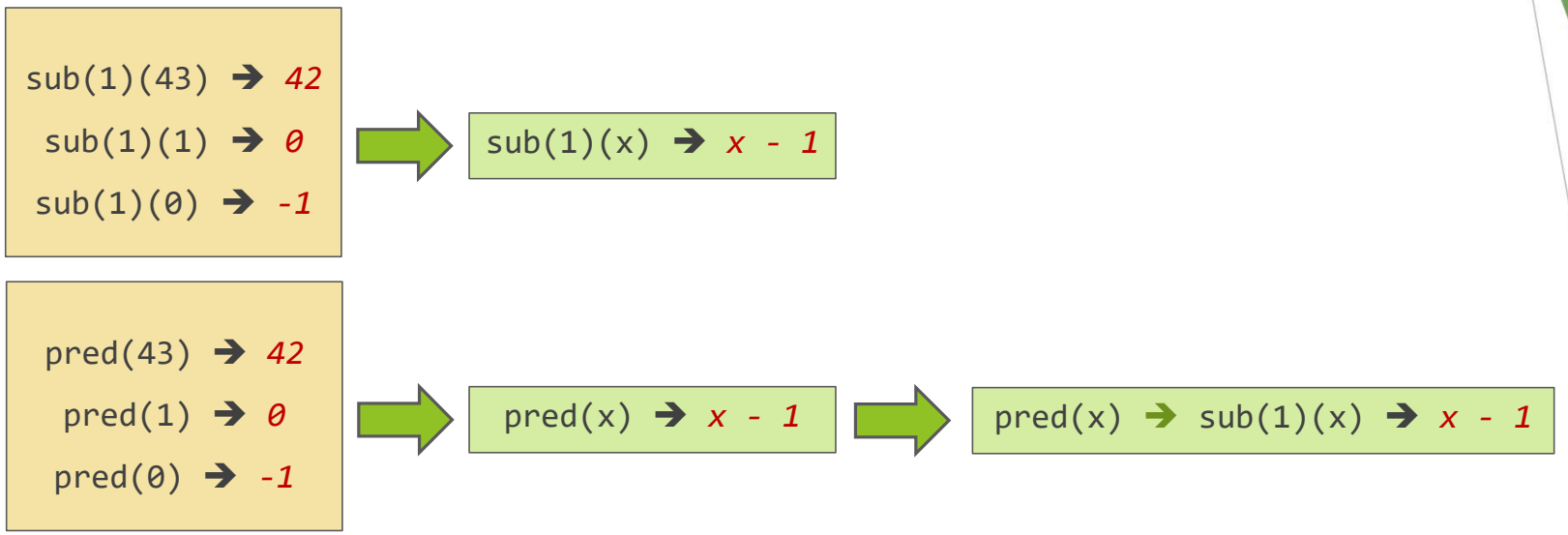
- ¿No sería genial tener una función llamada ***pred*** que calcula el predecesor de un número?
  - Sólo requiere un argumento
  - El nombre deja claro su propósito
- Sí, pero no podemos decirle a *Zilly* directamente: "dame una función ***pred***, tal que ***pred(x)*** ➔ ***x - 1***" …
  - … y tampoco "dame una función ***pred*** que calcula el predecesor"... Zilly no es IA
- Necesitamos definir ***pred*** en términos de algo que la máquina virtual ya entiende, y a eso vamos en el paso 2

Functional Abstraction: defining functions
step 2: define a parameterized rewriting rule



- This is easier than it sounds
  - Particularly if you started the functional abstraction journey with a known expression ☺
- In the current case, the expression is ***sub(1)(x)*** … we already know that it computes *x – 1*
  - To evaluate ***sub(1)(x)*** we only need a value for the x … (and a VM that can perform the subtraction, of course!)
  - Which is also the only piece of information our, *still unimplemented*, ***pred(x)*** needs.
  - The expression ***pred(x)*** rewriting rule is very simple: *pred(x)* ➔ *sub(1)(x)*
  - Note our use of the green arrow ➔ to denote the expression rewriting and the black arrow ➔ to denote evaluation

Abstracción funcional: definición de funciones
paso 2: definir una regla de reescritura parametrizada



- Esto es más fácil de lo que parece
  - Sobre todo, si comenzaste el proceso de abstracción funcional con una expresión conocida ☺
- En el caso actual, la expresión es sub(1)(x) … ya sabemos que calcula *x – 1*
  - Para evaluar ***sub(1)(x)***, sólo necesitamos un valor para x … (y una VM que sabe computar la resta, ¡por supuesto!)
  - Esta es también la única información que necesita nuestra función pred(x), aún sin implementar
  - La regla de reescritura de la expresión ***pred(x)*** es muy simple: *pred(x)* ➔ *sub(1)(x)*
  - Note el uso de la flecha verde ➔ para denotar la reescritura de la expresión y la flecha negra ➔ para denotar la evaluación

Functional Abstraction: defining functions

step 3: define the abstraction, expressing the rewriting rule with an anonymous function

```
sub(1)(43) ➔ 42
sub(1)(1) ➔ 0
sub(1)(0) ➔ -1
```

```
sub(1)(x) ➔ x - 1
```

```
pred(43) ➔ 42
pred(1) ➔ 0
pred(0) ➔ -1
```

```
pred(x) ➔ x - 1
```

```
pred(x) ➔ sub(1)(x) ➔ x - 1
```

The critical moment has arrived: here is the definition of the predecessor function pred in Zilly!

```
Z => Z pred := fn(Z x) -> sub(1)(x);
```

Which you can literally (and pedantically) read as follows: “I am defining a variable of type Z => Z named ***pred***, …
… initializing it (:=) with a function (fn) that, given an integer x (Z x), computes (->) sub(1)(x)”

You can’t deny it: this ***is a variable definition*** of the form <*type*> <*symbol*> := <*initializer*> ;
And remember: the <*initializer*> is an <*expression*> … the implications may surprise you.

Abstracción funcional: definición de funciones

paso 3: definir la abstracción, expresando la regla de reescritura con una función anónima

```
sub(1)(43) ➔ 42
sub(1)(1) ➔ 0
sub(1)(0) ➔ -1
```

➔ `sub(1)(x) ➔ x - 1`

```
pred(43) ➔ 42
pred(1) ➔ 0
pred(0) ➔ -1
```

➔ `pred(x) ➔ x - 1` ➔ `pred(x) ➔ sub(1)(x) ➔ x - 1`

Ha llegado el momento crítico: ¡aquí está la definición de la función predecesora pred en Zilly!

```
Z => Z pred := fn(Z x) -> sub(1)(x);
```

Que pueden *literalmente* (y con meticulosidad) leer así: "Estoy definiendo una variable de tipo Z => Z llamada ***pred***, …
… inicializándola (:=) con una función (fn) que, dado un entero x (Z x) , calcula (->) sub(1)(x)"

No puedes negarlo: esta es una ***definición de variable*** de la forma <*tipo*> <*símbolo*> := <*inicializador*> ;
Y recuerden que el <*inicializador*> es una <*expresión*> … las implicaciones pueden sorprenderte.

Functional Abstraction and Function Application

The last step is not as brutal as it may seem at first: we went from pred(x) → sub(1)(x) to pred = (x) → sub(1)(x) …
… only adding type declarations (e.g.: Z => Z) and some extra syntactic sugar (fn, := , and ;)

Type declarations could be inferred (an advanced topic) but stating types explicitly has some advantages.
What matters is that ***pred*** is just a variable *initialized* with a function! There are *five concepts* worth remembering here:

- `Z => Z pred := fn(Z x) -> sub(1)(x);` is a *variable definition* - the variable is of type **`Z => Z`**, a function type!
- `fn(Z x) -> sub(1)(x)` is an *anonymous function*, better known as a *lambda expression*
- `pred(43)` is a *function application* and `(fn(Z x) -> sub(1)(x))(43)` is a *function application* (an identical one in this case)
- Functions are 1st class values: you can assign them to variables, pass them as arguments, and return them as results
- Functions are values that can be applied to values to compute values

Under the leadership of your theory professor, you will learn the history and importance of functional abstraction, function application, and function composition. Run the code below, and if you learned something new today, let us know in a short email.

```
sys.reset();

Z => Z pred := fn(Z x) -> sub(1)(x);

pred

fn(Z x) -> sub(1)(x)

pred(43)

(fn(Z x) -> sub(1)(x))(43)
```

Abstracción Funcional y Aplicación de Funciones

El último paso no es tan drástico como podría parecer a primera vista: pasamos de pred(x) → sub(1)(x) a pred = (x) → sub(1)(x) …
… añadiendo sólo declaraciones de tipo (por ejemplo: Z => Z) y algo de azúcar sintáctico adicional (fn, := , and ;)

Las declaraciones de tipo se pueden inferir (tema avanzado), pero indicar los tipos explícitamente tiene algunas ventajas.
Lo importante es que pred es simplemente una variable inicializada con una función. Hay *cinco conceptos* que recordar:

- `Z => Z pred := fn(Z x) -> sub(1)(x);` es una *definición de variable* – la variable es de tipo **`Z => Z`**, ¡un tipo función!
- `fn(Z x) -> sub(1)(x)` es una *función anónima* , mejor conocida como *expresión lambda*
- `pred(43)` es una *aplicación de función* y `(fn(Z x) -> sub(1)(x))(43)` es una *aplicación de función* (idéntica en este caso)
- Las funciones son valores de primera clase: se pueden asignar a variables, pasar como argumentos, y devolver como resultados
- Las funciones son valores que se pueden aplicar a valores para calcular valores

Bajo la guía de su profesor de teoría, aprenderán la historia e importancia de la abstracción funcional, la aplicación funcional, y la composición funcional. Ejecuten el código a continuación y, si aprendieron algo hoy, Háganoslo saber en un correo electrónico.

```
sys.reset();

Z => Z pred := fn(Z x) -> sub(1)(x);

pred

fn(Z x) -> sub(1)(x)

pred(43)

(fn(Z x) -> sub(1)(x))(43)
```

Going the distance: functional abstraction with two variables

- Remember this beautiful "long sausage"? (well, not that long) ☺

  if(lt(y)(x), x, y)

- Step 0: "long sausage" detected [check] … opportunity for functional abstraction activated!
- Step 1: meaningful abstraction ***min(x, y)*** desired … but …
  … functions in Zilly are mono-argument, so ***min(x)(y)*** it is! [check]
- Step 2: parameterized rewriting rule ***min(x)(y)*** ➔ ***if(lt(y)(x), x, y)*** [check]
- Step 3: defining abstraction expressing the rewriting rule with a lambda expression, in *Zilly* sugar
  Step 3 a: separate name: ***fn(x) -> fn(y) -> if(lt(y)(x), x, y)*** [lambda expression check]
  Step 3 b: declare parameter types: ***fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y)*** [typed lambda check]
  Step 3 c: add symbol and terminator: ***min := fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y);*** [var def incomplete]
  Step 3 d: add type declaration: ***Z => Z => Z min := fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y);*** [check]

- Result: ***Z => Z => Z min := fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y);***
- Verify that ***min*** is working as you expect! Also try this: ***min(42)(min(21)(min(8)(67)))***
- Is the above hard? Not really, producing the expression ***if(lt(y)(x), x, y)*** requires more reasoning.
- And that's what you are going to do in the quiz: use ***reasoning*** to define functional abstractions.

Llegando al final: abstracción funcional con dos variables

- ¿Recuerdan este bello "chorizo largo"? (bueno, no tanto) ☺

  if(lt(y)(x), x, y)

- Paso 0: "chorizo largo" detectado [check] … ¡oportunidad de abstracción funcional activada!
- Paso 1: abstracción con sentido deseada ***min(x, y)*** … pero …
  … las funciones en *Zilly* solo toman un argumento: ¡vamos con ***min(x)(y)***! [check]
- Paso 2: regla de reescritura parametrizada ***min(x)(y)*** ➔ ***if(lt(y)(x), x, y)*** [check]
- Paso 3: definir la abstracción que expresa la reescritura con una expresión lambda, con azúcar de *Zilly*
  Paso 3a: separar el nombre: ***fn(x) -> fn(y) -> if(lt(y)(x), x, y)*** [expresión lambda check]
  Paso 3b: declarar los tipos de los parámetros: ***fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y)*** [typed lambda check]
  Paso 3c: añadir símbolo y terminador: ***min := fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y);*** [defvar incompleta]
  Paso 3d: añadir declaración de tipo: ***Z => Z => Z min := fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y);*** [check]

- Resultado: ***Z => Z => Z min := fn(Z x) -> fn(Z y) -> if(lt(y)(x), x, y);***
- Verifique que ***min*** funciona bien. Pruebe esta expresión también: ***min(42)(min(21)(min(8)(67)))***
- ¿Es lo anterior difícil? No tanto: formar la expresión ***if(lt(y)(x), x, y)*** requiere más razonamiento.
- Eso es lo que harán en el quiz: razonar para definir abstracciones funcionales.

Exercise - Define the following functions in Zilly

Logical operators (logical connectives) https://en.wikipedia.org/wiki/Propositional_logic

- *not(p)* ➔ true if p is false, false otherwise
  https://en.wikipedia.org/wiki/Negation
- *and(p)(q)* ➔ true if p and q are true, false otherwise
  https://en.wikipedia.org/wiki/Logical_conjunction
- *or(p)(q)* ➔ true if p or q (or both) are true, false otherwise
  https://en.wikipedia.org/wiki/Logical_disjunction

Comparison operators

- *le(n)(k)* ➔ $k \leq n$ (true if k is less than or equal to n, false otherwise)
- *eq(n)(k)* ➔ $k = n$ (true if k is equal to n, false otherwise)
- *ne(n)(k)* ➔ $k \neq n$ (true if k is different from n, false otherwise)
- *ge(n)(k)* ➔ $k \geq n$ (true if k is greater than or equal to n, false otherwise)
- *gt(n)(k)* ➔ $k > n$ (true if k is greater than n, false otherwise)

- *max(x)(y)* ➔ x if x is greater than y, y otherwise
- *add(n)(k)* ➔ $k + n$
  - Tip: implement *chs(n)* ➔ - n first, then use *sub* to implement *add*
    the name *chs* comes from *change sign* ☺

Ejercicio - Defina las funciones siguientes en Zilly

Operadores lógicos (conectores lógicos) https://es.wikipedia.org/wiki/Lógica_proposicional

- *not(p)* ➔ verdadero si p es falso, falso en caso contrario
  https://es.wikipedia.org/wiki/Negación_lógica
- *and(p)(q)* ➔ verdadero si p y q son verdaderos, falso en caso contrario
  https://es.wikipedia.org/wiki/Conjunción_lógica
- *or(p)(q)* ➔ verdadero si p o q (o ambos) son verdaderos, falso en caso contrario
  https://es.wikipedia.org/wiki/Disyunción_lógica

Operadores de comparación

- *le(n)(k)* ➔ $k \leq n$ (verdadero si k es menor o igual a n, falso en caso contrario)
- *eq(n)(k)* ➔ $k = n$ (verdadero si k es igual a n, falso en caso contrario)
- *ne(n)(k)* ➔ $k \neq n$ (verdadero si k es diferente de n, falso en caso contrario)
- *ge(n)(k)* ➔ $k \geq n$ (verdadero si k es mayor o igual a n, falso en caso contrario)
- *gt(n)(k)* ➔ $k > n$ (verdadero si k es mayor que n, falso en caso contrario)

- *max(x)(y)* ➔ x si x es mayor que y, y en caso contrario
- *add(n)(k)* ➔ $k + n$
  - Tip: implement *chs(n)* ➔ - n first, then use *sub* to implement *add*
    the name *chs* comes from *change sign* ☺