LE.P.I.F.Y.O 🎯

Lenguaje Para Introducción a Funcional y Objetos

🚀 Introducción

El objetivo de este trabajo práctico es aplicar conceptos que tienen que ver con sistemas de tipado estático y la combinación de herramientas provenientes de los paradigmas de objetos y funcional para resolver un problema concreto.

Por lo tanto, no sólo forma parte del TP poder resolver la funcionalidad pedida, sino también utilizar las distintas herramientas y experimentar con ellas, en lugar de sólo utilizar las que ya conocen. Por supuesto, la idea es que analicen las ventajas y desventajas en cada caso, pudiendo justificar así las decisiones de diseño que tomen.

Les recomendamos entonces no intentar diseñar una solución de entrada sólo con objetos. En lugar de eso, analicen en qué lugares sería más simple o conveniente el uso de pattern-matching, funciones de orden superior, etc.

💡**Nota: antes de empezar, asegúrense de haber leído el enunciado completo del TP.**

🏛️ El dominio del problema

Durante el TP vamos a estar construyendo un lenguaje de programación imperativo simple, llamado “Lepifyo” (*Lenguaje Para Introducción a Funcional y Objetos*).

Como el dominio de nuestro problema va a ser un lenguaje de programación, hay que tener en cuenta que los elementos de nuestro programa no van a representar cosas como un Arma o un Guerrero, sino que van a representar conceptos y estructuras propias del lenguaje de programación que estamos implementando: por ejemplo una variable, un número, una operación de suma, etc. Esto es similar a lo que sucedió durante el TP 1. En las siguientes secciones vamos a hablar sobre el alcance del TP en cuanto a las funcionalidades del lenguaje.

📜 Especificación del lenguaje

Lepifyo va a ser un lenguaje imperativo simple dinámicamente tipado, basado en texto[1]. Los programas van a tener variables, asignaciones, operaciones y lambdas. Una vez que alguien escribe un programa en Lepifyo, queremos poder realizar algunas operaciones, como:

- Ejecutar el programa (¡por supuesto!).
- Realizar un análisis estático sobre el código, en búsqueda de posibles problemas.

Ahora bien, ¿cómo encaramos el desarrollo de un lenguaje de programación? Esto es, en general, una tarea compleja, por lo que usualmente se divide el problema en partes.

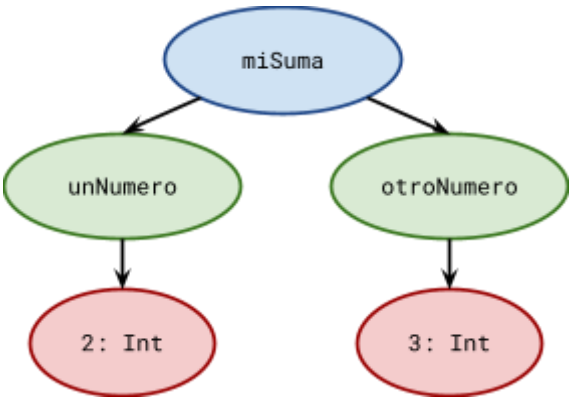
Dividiendo el problema

Para poder ejecutar o analizar un programa, tenemos que interpretar de alguna manera el arreglo de caracteres que representa al mismo. Notemos que:

- Poder entender qué elementos del programa representa cada porción de texto (es decir, saber si lo que escribí es una suma, una asignación u otra cosa) es algo que vamos a necesitar tanto para ejecutar el programa como para analizarlo.
- Manipular el programa a mano como un arreglo de caracteres es bastante incómodo.

En consecuencia, nos gustaría tener un modelo que represente al programa, independientemente de la operación que queremos realizar. Esto nos lleva a dividir el modelo en dos partes:

1. El *parser*[2] o analizador sintáctico, que se va a encargar de reificar al programa en base al texto. Es decir, el parser recibirá el texto del programa, y devolverá un conjunto de objetos que representan al programa. A este resultado vamos a llamarlo "árbol de sintaxis" o *AST*[3].
   Por ejemplo, un diagrama de objetos para el AST correspondiente a la expresión `2 + 3` podría ser algo como:

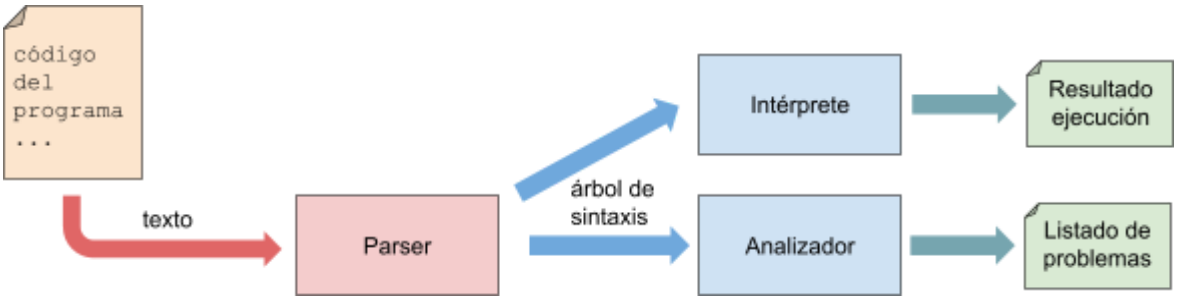
   

   Forma parte del TP el desarrollo de los objetos que conforman el AST, no así el desarrollo del parser.

---

[1] Es decir, los programas van a estar representados mediante texto (como en la mayoría de los lenguajes de programación).
[2] Parser: analiza una cadena de símbolos de acuerdo a las reglas de una gramática formal y la transforma en un AST.
[3] AST o Abstract Syntax Tree: es el modelo de objetos o estructura que resulta de parsear un programa, y que representa la estructura del mismo.

2. Los componentes que vamos a usar para realizar operaciones sobre el programa (ejecutar y analizar estáticamente), que van a tener como entrada al AST en lugar del texto.



📐 Alcance del TP

Van a tener a disposición una implementación del parser para que puedan usarlo con el AST que definan. El parser estará incluido en el repo del assignment.

Por lo tanto, el alcance del TP sólo incluye:

- El diseño de una estructura conveniente para el AST.
- El desarrollo del intérprete y el analizador.

Queda fuera del alcance del TP hacer cualquier modificación al parser, o el desarrollo de cualquier otra cosa extra que podamos imaginarnos, como por ejemplo un IDE.

Las operaciones sobre los programas

Como ya mencionamos anteriormente, vamos a trabajar sobre algunas operaciones que pueden realizarse sobre un programa:

- **Análisis estático del código**: consiste en recorrer el AST, analizándolo en base a ciertas reglas. El resultado del análisis es una lista de problemas detectados.

  Por ejemplo, una regla de análisis puede servir para detectar divisiones por cero explícitas. Entonces, si como parte del programa tenemos una división como `3 / 0`, el resultado del análisis debería incluir un error en referencia a esa expresión.
  👁️ Ojo, esto no involucra ejecutar ni la división ni el programa, la idea es sólo analizar la estructura del mismo (es decir, los elementos del AST) y detectar problemas.

  Queremos que el analizador sea "*configurable*", es decir, que sus reglas no sean fijas, sino que podamos pasarle al analizador el conjunto de reglas que queremos que utilice. La idea es que el analizador sea como un *"motor de reglas"*, que tiene la lógica para recorrer el AST evaluando las reglas proporcionadas.

  ⚠️ *Importante*: también queremos tener la posibilidad de definir reglas de análisis nuevas, más allá de las que implementemos como parte del TP.
  📄 Necesitamos especificar los pasos que habría que seguir para poder implementar una nueva regla, incluyendo un ejemplo. Pueden hacerlo en un archivo en el repo, en

formato markdown. Les recomendamos que hagan esto al final, ya que la interfaz probablemente vaya cambiando durante el transcurso del desarrollo del TP.

- **Ejecución**: consiste en recorrer el AST, pero esta vez para ejecutar el programa[4].

  Por ejemplo, si nuestro programa consiste sólo en la expresión `2 + 3`, al ejecutarlo debería dar 5 como resultado. Forma parte del TP definir la representación de los valores en runtime.

Estructura del TP

Para poder implementar todo esto de manera iterativa e incremental, en las siguientes secciones presentamos una descripción de los requisitos para el AST y cada uno de los componentes.

En cada sección vamos a describir los distintos tipos de elementos que se tienen que considerar para el AST, y qué se debería considerar para cada una de las operaciones a realizar (ejecución y análisis). De todos modos, les recomendamos leer el TP en su totalidad antes de empezar a trabajar.

🔣 Booleanos, números enteros y operaciones

El objetivo es modelar los booleanos, números enteros y las operaciones entre ellos.

AST

Tenemos que decidir qué objetos vamos a usar para representar:

- Números enteros literales que se hayan incluído en el programa.
- Booleanos literales `true` y `false`.
- Operaciones entre números enteros, que son:
  - **Aritméticas**: suma, resta, multiplicación y división[5]. Tendrán como resultado a otro número entero.
  - **De comparación**: igual, distinto, menor, mayor, menor o igual, mayor o igual. Tendrán como resultado a un booleano.

Operaciones sobre los programas

Análisis

Recordemos que, para poder realizar esta operación sobre un programa tendremos que desarrollar un analizador que reciba el AST de un programa, evalúe ciertas reglas, y nos de como resultado una lista de problemas encontrados.

Entonces, la salida del analizador será una lista de problemas encontrados. Un problema tendrá:

- Una descripción. Por ejemplo: “No se puede dividir por cero”.

---

[4] Usualmente se denomina “intérprete” al componente que ejecuta el programa en base a su AST.
[5] División entera, que va a devolver el cociente del resultado de la división, ignorando el resto.

- Un nivel de gravedad: Error o Advertencia.
- El elemento del AST que tiene el problema, por ejemplo `3 / 0`.

Las reglas a implementar tienen que ver con los números enteros y las operaciones entre ellos, a saber:

- **División por cero**: queremos que el analizador reporte un error cuando tenemos una división por el literal 0.
  Por ejemplo, el siguiente programa debería generar un error al ser analizado: `3 / 0`.
  Sin embargo, como no estamos ejecutando el programa, la detección de divisiones por cero que no estén explícitas queda fuera del alcance. Por ejemplo, el siguiente programa no va a generar un error al ser analizado: `3 / (2 - 2)`.

- **Operaciones redundantes**: algunas operaciones entre números son consideradas redundantes, por ejemplo sumar 0 o dividir por 1. Queremos que el analizador genere una advertencia si en el programa se realizan algunas de estas operaciones redundantes.
  Las operaciones redundantes que vamos a considerar son: restar 0, sumar 0, multiplicar por 1, dividir por 1. En el caso de la suma y la multiplicación, al ser operaciones conmutativas, consideramos la operación como redundante si esos casos se producen a izquierda o derecha.
  Por ejemplo, el siguiente programa debería generar una advertencia al ser analizado: `3 / 1`.
  Al igual que el caso anterior, la detección de operaciones redundantes que no estén explícitas queda fuera del alcance. Por ejemplo, el siguiente programa no va a generar un error al ser analizado: `3 / (5 - 4)`.

Ejecución

Tendremos que desarrollar un intérprete, que reciba el AST de un programa (hasta este momento, consistente en números, booleanos, operaciones aritméticas y de comparación), y devuelva el resultado de haberlo ejecutado.

No incluimos aquí una especificación de los resultados que genera la ejecución de cada una de las operaciones aritméticas y de comparación que listamos en los puntos anteriores: la ejecución deberá producir como resultado los valores usuales que corresponden a estas operaciones. Sin embargo, si hay alguna duda puntual no duden en consultar.

🤦 REPL

Vamos a implementar una herramienta para desarrolladores: el REPL. Al iniciarlo, el usuario debe ver un *prompt* en la consola:

```
> _
```

Al ingresar expresiones y presionar <*Enter*>, se deberán ejecutar, mostrar el resultado, y mostrar nuevamente el *prompt*:

```
> 1 + 1
2
> _
```

Para salir del REPL, se deberá ingresar el comando `exit`.

✍ Para obtener entrada del usuario, se puede usar el método `readLine` de Scala.

📦 Variables

Vamos a agregar variables a nuestro lenguaje.

AST

Al agregar variables a nuestro lenguaje tenemos que considerar a las operaciones que queremos realizar con ellas:

- **Declaración de variables**: la declaración de una variable incluye su nombre y valor inicial:

  ```
  let edad = 27
  ```

- **Lectura**: una variable puede utilizarse en nuestro programa si la referenciamos. Por ejemplo, consideremos el siguiente programa, que hace que el valor inicial de `añoNacimiento` sea `añoActual - edad`:

  ```
  let añoActual = 2021
  let edad = 25
  let añoNacimiento = añoActual - edad
  ```

  En este caso, la resta es entre dos variables (en lugar de ser entre dos enteros literales).

  Obviamente, se pueden combinar valores y variables en una misma expresión:

  ```
  edad + 1
  ```

- **Escritura**: las variables pueden ser *asignadas* (para modificar su valor). Por ejemplo, el siguiente programa declara la variable `añoActual`, y luego modifica su valor:

  ```
  let añoActual = 2020
  añoActual = añoActual + 1
  ```

Vemos que al considerar la existencia de variables, tiene sentido que un programa consista en una *secuencia* de instrucciones.

Operaciones sobre los programas

Análisis

- Tenemos que detectar cuando una variable está duplicada (es decir, cuando se declaran dos variables con el mismo nombre). Por ejemplo, el siguiente programa debería generar un problema de este tipo, ya que la variable `añoActual` se declara dos veces:

```
let añoActual = 2020
let añoActual = 2021
```

Ejecución

Debemos poder ejecutar programas que declaren, usen y asignen variables.

Forma parte del TP decidir cuál va a ser la semántica del uso de variables (esto incluye la consideración de casos bordes).

Interacción con el REPL

Si se definen variables, estas deberían ser accesibles desde el REPL:

```
> let x = 1
> x
1
> _
```

λ Expresiones Lambda

AST

Esto tiene dos partes: por un lado la declaración de una lambda, y por el otro su aplicación.

Declaración de lambda

Para declarar una lambda, especificamos el nombre de sus parámetros y el cuerpo del lambda. Por ejemplo, una lambda muy simple:

```
let sumarDos = (n) -> n + 2
```

Esto declara una lambda con un parámetro “n”, y la asigna en la variable `sumarDos`.

Podemos tener varios o ningún parámetro:

```
let sumar = (a, b) -> a + b
let elSiete = () -> 7
```

Y también podemos tener más de una instrucción en el cuerpo de la lambda:

```
let sumarDos = (n) -> {
    let y = 2
    n + y
}
```

Aplicación de lambda

La aplicación de una lambda involucra pasarle los argumentos correspondientes a sus parámetros. Para nuestro ejemplo anterior, un uso válido sería:

```
let resultado = sumarDos(3)
```

También, por supuesto, se puede aplicar un lambda sin necesidad de usar una variable. Lo siguiente es equivalente al ejemplo anterior:

```
let resultado = ((n) -> n + 2)(3)
```

Análisis

- Se debería generar un problema (ya sea una advertencia o un error, a criterio del grupo) cuando la última instrucción de una lambda no es una expresión. Por ejemplo, el siguiente programa debería generar un problema:

  ```
  let sumarDos = (n) -> { let y = 2 }
  ```

- Detectar cuando una lambda tiene más de un parámetro con el mismo nombre. Por ejemplo, lo siguiente debería generar un problema:

  ```
  ((n, n) -> n)
  ```

Ejecución

Implementar la declaración y aplicación de lambdas.

Al ejecutarse, una lambda devolverá el resultado de la última colaboración definida en su cuerpo. Por ejemplo, el siguiente código producirá `5` como resultado:

```
let sumarDos = (n) -> {
    let y = 2
    n + y
}
```

```
sumarDos(3)
```

Una lambda es un *closure*[6]. Esto significa que va a generar un contexto propio para sus variables locales, pero se pueden referenciar y modificar las variables del contexto en el que se definió la lambda. A continuación incluimos algunos ejemplos sobre lo que esto implica.

- Se pueden leer las variables del contexto en el que se definió la lambda:

  ```
  let x = 1

  let devolverX = () -> x
  ```

  `devolverX()` debería ser 1

- Los parámetros de la lambda ocultan a las variables del contexto en el que se la definió:

  ```
  let x = 1

  let devolverX = (x) -> x
  ```

  `devolverX(2)` debería ser 2, `x` debería seguir siendo 1

- Se pueden modificar las variables del contexto en el que se definió la lambda:

  ```
  let x = 1

  let modificarX = (y) -> {
    x = y
  }
  modificarX(2)
  ```

  `x` debería ser 2

- Se pueden definir y modificar variables locales dentro de la lambda:

  ```
  let devolverY = () -> {
    let y = 1
    y = y + 1
    y
  }
  ```

  `devolverY()` debería ser 2, no debería haber una variable `y` definida luego de la ejecución de la lambda.

- Las variables locales ocultan a las variables del contexto en el que se definió la lambda:

  ```
  let x = 1

  let devolverX = () -> {
  ```

[6] Que sea un closure quiere decir que le vamos a asociar un contexto en forma *lexica*; es decir, según en donde fue *definida* la lambda, y según donde sea *aplicada* (que sería asociarlo en forma dinámica).

```
  let x = 2
  x = x + 1
  x
}
```

`devolverX()` debería ser 3, `x` debería seguir siendo 1.

- Todos los ejemplos anteriores aplican de la misma manera al tener lambdas anidadas. Cada lambda generará su contexto, teniendo al contexto en donde fue definida como “contexto padre”:

```
let x = 1

let sumar = (x) -> (y) -> x + y
```

`sumar(2)(x)` debería ser 3

```
let x = 1
let devolverX = () -> x
let otroDevolverX = () -> {
  let x = 2
  devolverX()
}
```

`otroDevolverX()` debería ser 1

- Se pueden acceder a las variables definidas en cualquier contexto que sea ancestro del contexto de la lambda que se aplica:

```
let x = 1

let sumarAX = (y) -> {
  ((y) -> x + y - 1)(y + 1)
}
```

`sumarAX(2)` debería ser 3

- Además, al aplicar un lambda, sus argumentos se evalúan de izquierda a derecha:

```
let x = 1

let modificarXYDevolverSucesor = (y) -> {
  x = y
  y + 1
}

let sumar = (x) -> (y) -> x + y

let aplicarA = (f, x) -> {
  f(x)
}
```

`aplicarA(sumar(modificarXYDevolverSucesor(2)), x)` debería ser 5

🍁 El parser

Uso

La clase `ParserLepifyo` es la encargada de generar instancias del parser.
Al crear una instancia del parser hay que indicarle el tipo del objeto que representa un Programa y el tipo de los objetos que representan a los elementos del AST. Además, tenemos que pasarle las funciones que usará para crear cada elemento del AST.

```
val programa = (expresiones: List[Expresion]) => ???
val numero = (n: Int) => ???
val booleano = (b: Boolean) => ???
val cadena = (c: String) => ???
val suma = (operando1: Expresion, operando2: Expresion) => ???
val resta = (operando1: Expresion, operando2: Expresion) => ???
val multiplicacion = (operando1: Expresion, operando2: Expresion) => ???
val division = (operando1: Expresion, operando2: Expresion) => ???
val igual = (operando1: Expresion, operando2: Expresion) => ???
val distinto = (operando1: Expresion, operando2: Expresion) => ???
val mayor = (operando1: Expresion, operando2: Expresion) => ???
val mayorIgual = (operando1: Expresion, operando2: Expresion) => ???
val menor = (operando1: Expresion, operando2: Expresion) => ???
val menorIgual = (operando1: Expresion, operando2: Expresion) => ???
val declaracionVariable = (nombre: String, valorInicial: Expresion) => ???
val variable = (nombre: String) => ???
val asignacion = (nombre: String, valorNuevo: Expresion) => ???
val concatenacion = (operando1: Expresion, operando2: Expresion) => ???
val lambda = (parametros: List[String], cuerpo: List[Expresion]) => ???
val aplicacion = (funcion: Expresion, argumentos: List[Expresion]) => ???

val parser = new ParserLepifyo[Programa, Expresion](
  programa = programa,
  numero = numero,
  booleano = booleano,
  string = cadena,
  suma = suma,
  resta = resta,
  multiplicacion = multiplicacion,
  division = division,
  igual = igual,
  distinto = distinto,
  mayor = mayor,
  mayorIgual = mayorIgual,
  menor = menor,
  menorIgual = menorIgual,
  declaracionVariable = declaracionVariable,
  variable = variable,
  asignacion = asignacion,
  concatenacion = concatenacion,
  lambda = lambda,
  aplicacion = aplicacion
)
```

Ejemplos

Números y booleanos literales

```
parser.parsear("12") == programa(List(numero(12)))

parser.parsear("true == false") ==
  programa(List(igual(booleano(true), booleano(false))))
```

Operaciones y comparaciones

```
parser.parsear("12 + 34 + 56") == programa(List(suma(suma(12, 34), 56)))

parser.parsear("12 - 34 * 56") == programa(List(resta(12, multiplicacion(34, 56))))

parser.parsear("12 + 15 == 56 - 21") ==
  programa(List(igual(suma(12, 15), resta(56, 21))))

parser.parsear("12 < 56 != 14 > 13") ==
  programa(List(distinto(menor(12, 56), mayor(14, 13))))
```

Programas con una expresión por línea

```
parser.parsear("1 + 2\n2 + 1") ==
  programa(List(suma(1, 2), suma(2, 1)))
```

Variables

```
parser.parsear("let variable = 12") ==
  programa(List(declaracionVariable("variable", 12)))

parser.parsear("numerador / denominador") ==
  programa(List(division(variable("numerador"), variable("denominador"))))

parser.parsear("numerador = 10") == programa(List(asignacion("numerador", 10)))
```

Lambda

```
parser.parsear("() -> 2 + 2") == programa(List(lambda(List(), List(suma(2, 2)))))

parser.parsear("(x, y) -> x") ==
  programa(List(lambda(List("x", "y"), List(variable("x")))))

parser.parsear("() -> {\n\tlet y = 1\n\ty\n}") ==
  programa(List(lambda(List(), List(declaracionVariable("y", 1), variable("y")))))

parser.parsear("(() -> 2)()") ==
  programa(List(aplicacion(lambda(List(), List(2)), List())))

parser.parsear("f(2)") == programa(List(aplicacion(variable("f"), List(2))))
```

🛸 Bonus

🐛 Debugger

Vamos a implementar un debugger para el lenguaje.

Para activarlo, vamos a tener una función `halt` predefinida a nivel global. Al ejecutar un programa, si se aplica esta función sin argumentos, se deberá pausar la ejecución del programa en ese momento, y se abrirá un REPL que permitirá inspeccionar y modificar el contexto de ejecución. Al escribir `continue`, se continúa con la ejecución del programa.

Por ejemplo, si tenemos el siguiente programa:

```
let x = 1

let sumar = (x) -> (y) -> x + y

let aplicarA = (f, x) -> {
  halt()
  f(x)
}

aplicarA(sumar(1), 2)
```

Al ejecutarse, podemos tener la siguiente sesión de debugging:

```
El debugger se ha activado!
> x
2
> f(3)
4
> continue
```

🚪 Entrada/Salida

Vamos a agregar la posibilidad de que nuestros programas tengan entrada/salida.

AST

Para poder soportar las operaciones de entrada/salida, tenemos que considerar los siguientes elementos nuevos en el AST:

- **Strings**: que representan una secuencia de caracteres.

  Esto incluye tanto la representación de strings literales: `"hola"`
  Como la operación de concatenación: `"hola" ++ " mundo"`

- **Funciones de I/O**: funciones que nos permiten ejecutar operaciones de entrada/salida. A saber:

  - Entrada: para solicitarle valores al usuario vamos a tener una familia de funciones estilo “prompt”, que reciben una expresión que representa un string y devuelven el valor que el usuario ingresó. El tipo del valor de retorno dependerá de la función que se use. Las funciones son `promptInt`, `promptString` y `promptBool` (para solicitar un número entero, un string o un booleano, respectivamente).

    Por ejemplo, el siguiente programa define una variable `añoActual`, cuyo valor será solicitado al usuario:

    ```
    let añoActual = promptInt("Ingrese el año actual: ")
    ```

  - Salida: para mostrarle valores al usuario vamos a tener una función `printLn`, que recibe una expresión y tiene el efecto de imprimir el valor de la expresión en la pantalla.

    Por ejemplo, el siguiente programa imprime “hola mundo”:

    ```
    printLn("hola mundo")
    ```

Operaciones sobre los programas

Análisis

Nuevas reglas:

- La función `printLn` no genera un resultado que nos interese, la usamos solamente para producir un efecto. Por esto queremos generar un problema si encontramos que estamos utilizando el valor de retorno de `printLn` dentro de alguna expresión.

  Por ejemplo, el siguiente programa genera un problema:

  ```
  printLn("hola mundo") + 2
  ```

- Usamos las funciones `prompt*` para poder pedirle un valor al usuario y obtenerlo como resultado. Queremos que se genere un problema cuando se ignora el valor de retorno al usar estas funciones.

  Por ejemplo, el siguiente programa genera un problema:

  ```
  promptInt("Ingrese un número")
  promptInt("Ingrese un número")
  ```

Ejecución

Las consideraciones a tomar en cuenta para ejecutar los programas que incluyan strings y funciones de entrada/salida son:

- Tanto la *concatenación* como la función `printLn` pueden recibir expresiones que no representen un string (es decir, que sean números enteros o booleanos). En este caso, al ejecutarse el programa deberán transformarse estos valores a strings de forma transparente (por ejemplo, el `1` se transforma en `"1"`, y `true` se transforma en `"true"`), antes de realizar la operación.
- `prompt*` y `printLn` deberán realizar las lecturas/escrituras a partir de la entrada/salida estándar que estén definidas en Scala al momento de ejecutar el programa.
  💡 Idea: pensar en cómo se podría testear esto de forma automatizada. Pueden considerar diferentes maneras de diseñar el evaluador para que esto sea más o menos fácil de testear.
- `printLn` deberá imprimir el resultado de haber convertido su argumento a string seguido de un salto de línea (`"\n"`).