**Carrera**: Ingeniería en Sistemas Computacionales

**Asignatura**: Programación Lógica y Funcional

**Grupo**: 7SA

**Alumno:** Novelo Cruz, Raúl Armin

**Matrícula:** E17081430

# Laboratorio 1 Fundamentos

## ¡Bienvenido a 15-150!

Mis TA son:

Sus Andrew IDs son:

Mi carta de la sección de laboratorio es:

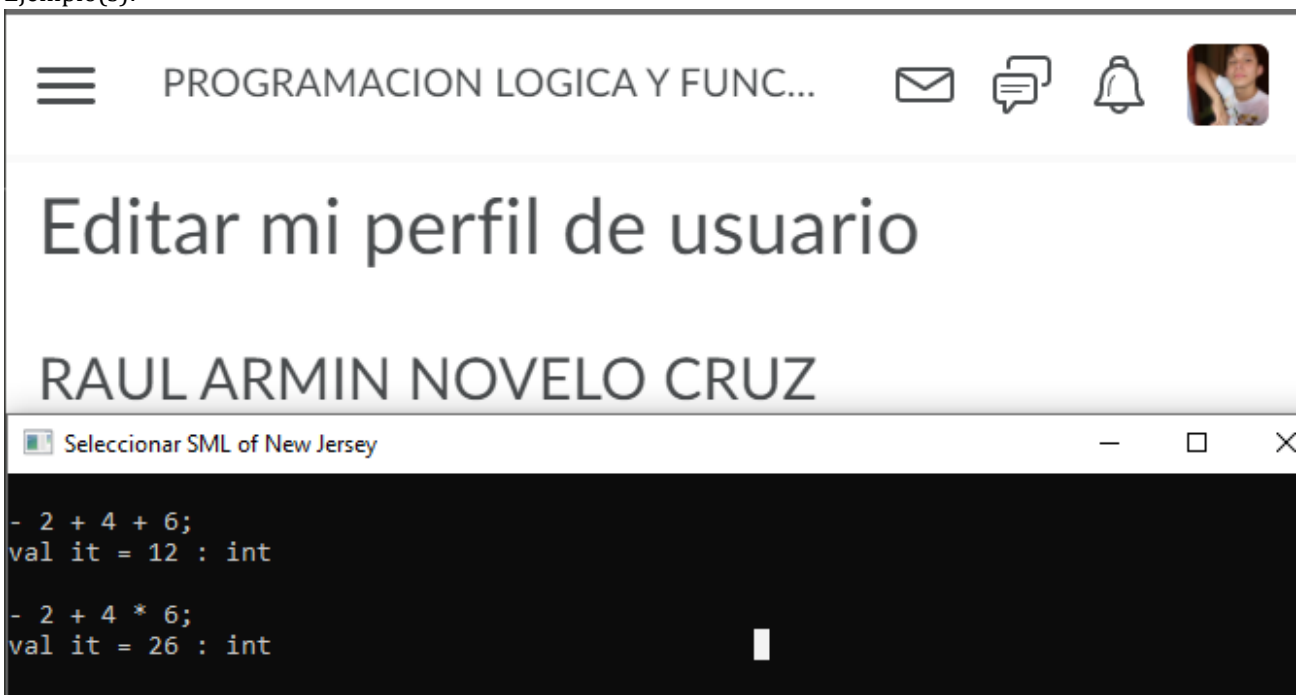
---

# 1 Conceptos básicos de SML

## 1.1 Expresiones

Las expresiones son la unidad básica de un programa SML.

Ejemplo(s):




## 1.2 Tipos

Un *tipo de* programa comprueba si no tiene errores de tipo; los errores de tipo proceden de expresiones que no están bien *tipadas*. Si tenemos:

e : t

Decimos que la expresión e tiene el tipo t.

Ejemplo(s):

También hay *tipos de función*. Un tipo de función se puede reconocer mediante una flecha en él, como:

e : t1 -> t2

Una expresión con este tipo tiene un *tipo* de argumento t1 y un tipo de *valor devuelto* de t2.

Ejemplo(s):

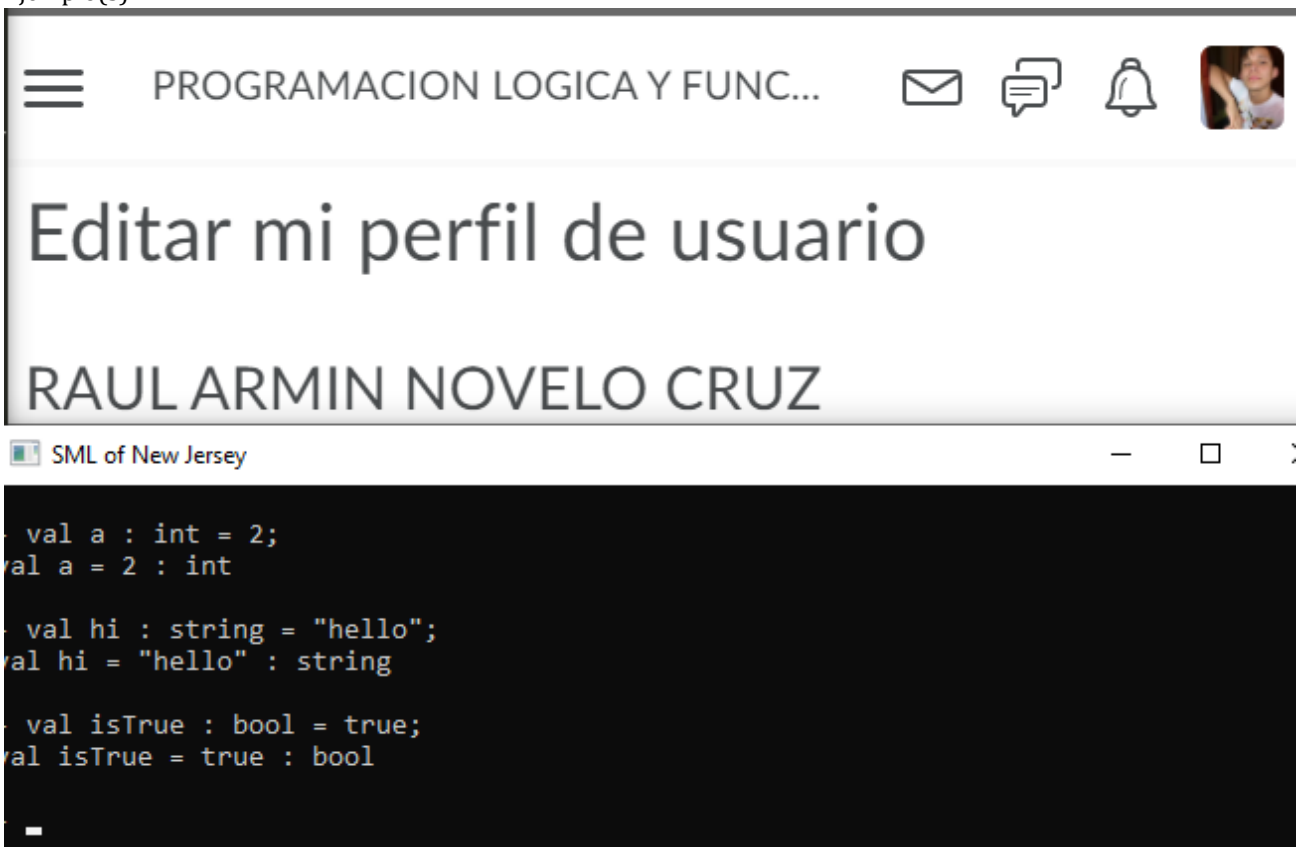
## 1.3 Declaraciones de val

Podemos enlazar valores a nombres de variables (también *llamados identificadores)*utilizando la sintaxis:

val <var> : <tipo> ?<expr>

Para que esto escriba la comprobación, la expresión <expr> debe tener el tipo <tipo>.

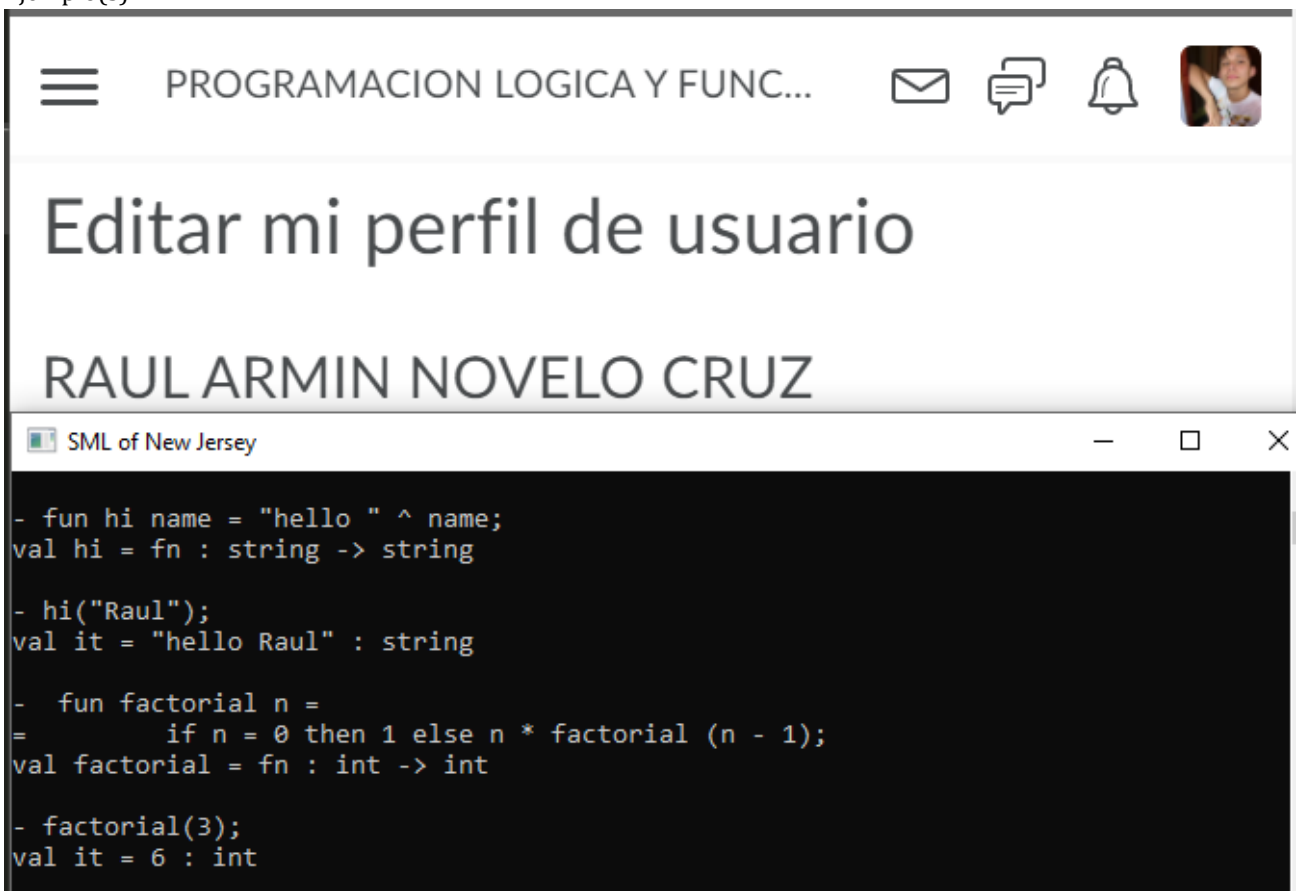
Ejemplo(s):




## 1.4 declaraciones divertidas

Las funciones de SML se pueden definir (o *declarar)*mediante la palabra clave fun. keyword.

Ejemplo(s):




## 1.5 Alcance

Las declaraciones tienen un atributo denominado *scope*, que está en todas partes donde se puede usar. Cuando se realiza una nueva declaración, esa declaración solo tiene acceso a los enlaces de nombre de variable creados *antes* de ella. Una declaración no se puede realizar a partir de enlaces creados *después de* ella; no está en el ámbito de dichos enlaces.

Usamos notación de enlace de entorno para ayudarnos a realizar un seguimiento de los enlaces. Tenga en cuenta que esta no es la sintaxis SML real. Anote lo siguiente con notación de enlace de entorno:

Val X: Int 10

Val y : Int = x * x
Val Z: Int = x + y

## 1.6 Sombreado

El enlace a nombres de variables en SML es diferente de la asignación en otros lenguajes de programación. Si se enlaza a la misma variable dos veces, la variable sigue enlazando el primer valor entre el primer y el segundo enlace; decimos que la segunda unión *sombrea* el primer enlace en lugar de reemplazarlo en todas partes, porque antes de la segunda unión el primer enlace todavía existe!

Ejemplo(s):

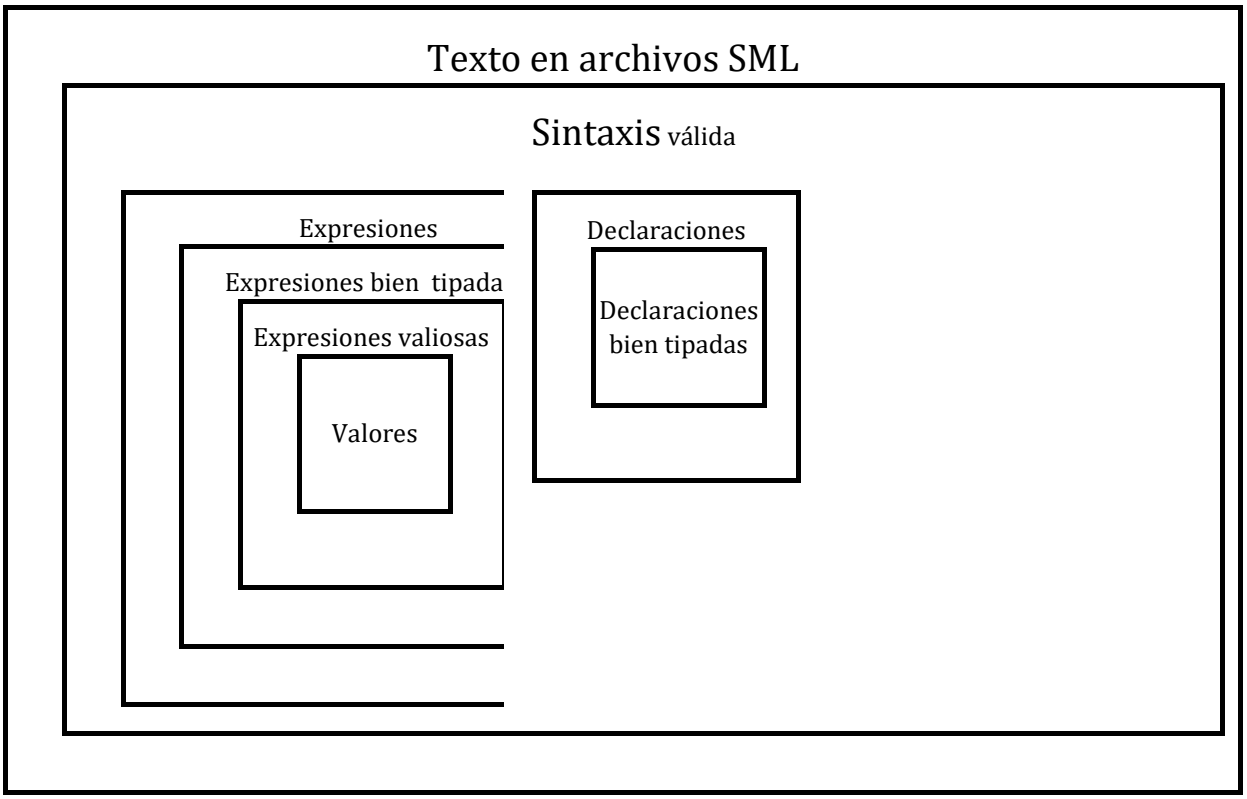



Figura 1: Una jerarquía simple de texto en SML.

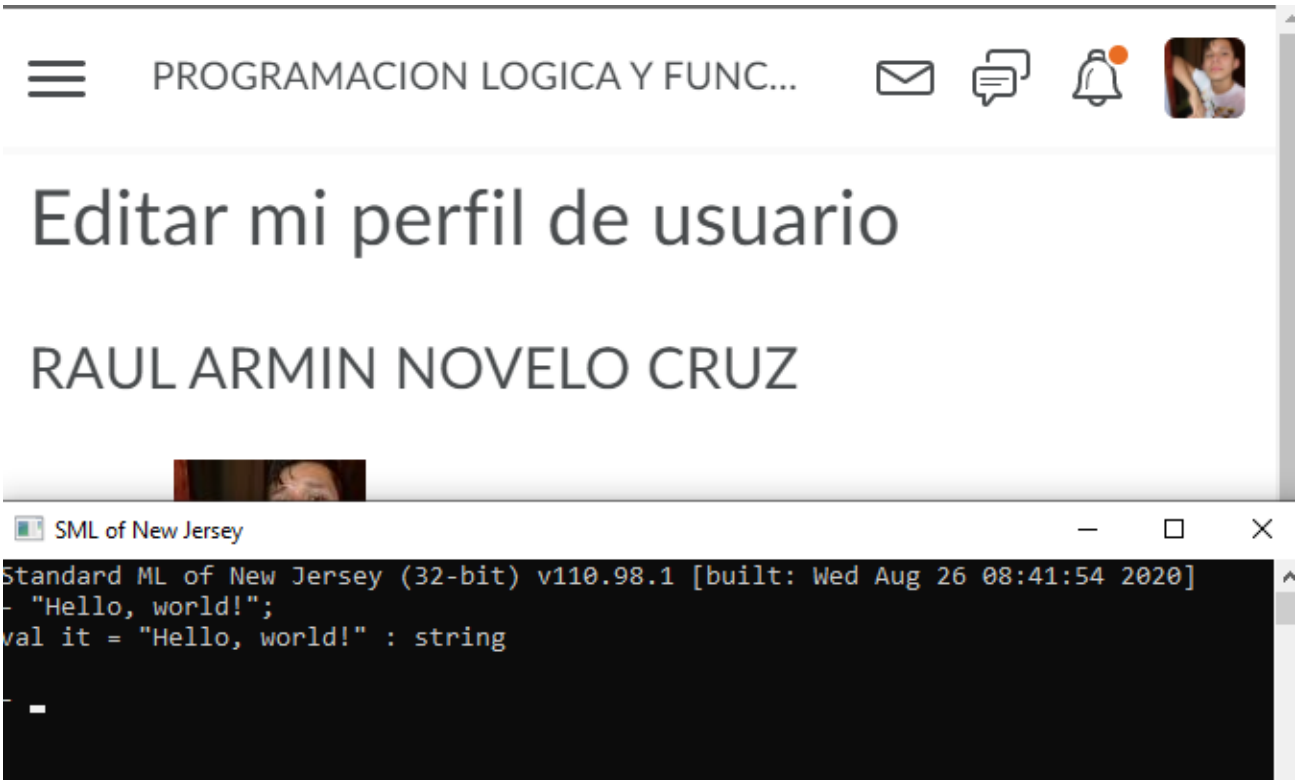
# 2 Introducción

Tiempo de configuración - echa un vistazo a este doc! Una vez que haya terminado, usted debe ser capaz de abrir el REPL usando smlnj y ejecutar

```
- "Hola", "mundo!" "world!" ;

val it "Hola, mundo!" : cadena
```

¡Asegúrate de que esto funcione! Pulse Ctrl-D para salir de la REPL una vez que haya terminado.


PROGRAMACION LOGICA Y FUNC...
Editar mi perfil de usuario
RAUL ARMIN NOVELO CRUZ


SML of New Jersey
Standard ML of New Jersey (32-bit) v110.98.1 [built: Wed Aug 26 08:41:54 2020]
- "Hello, world!";
val it = "Hello, world!" : string
-

# 3 Expresiones

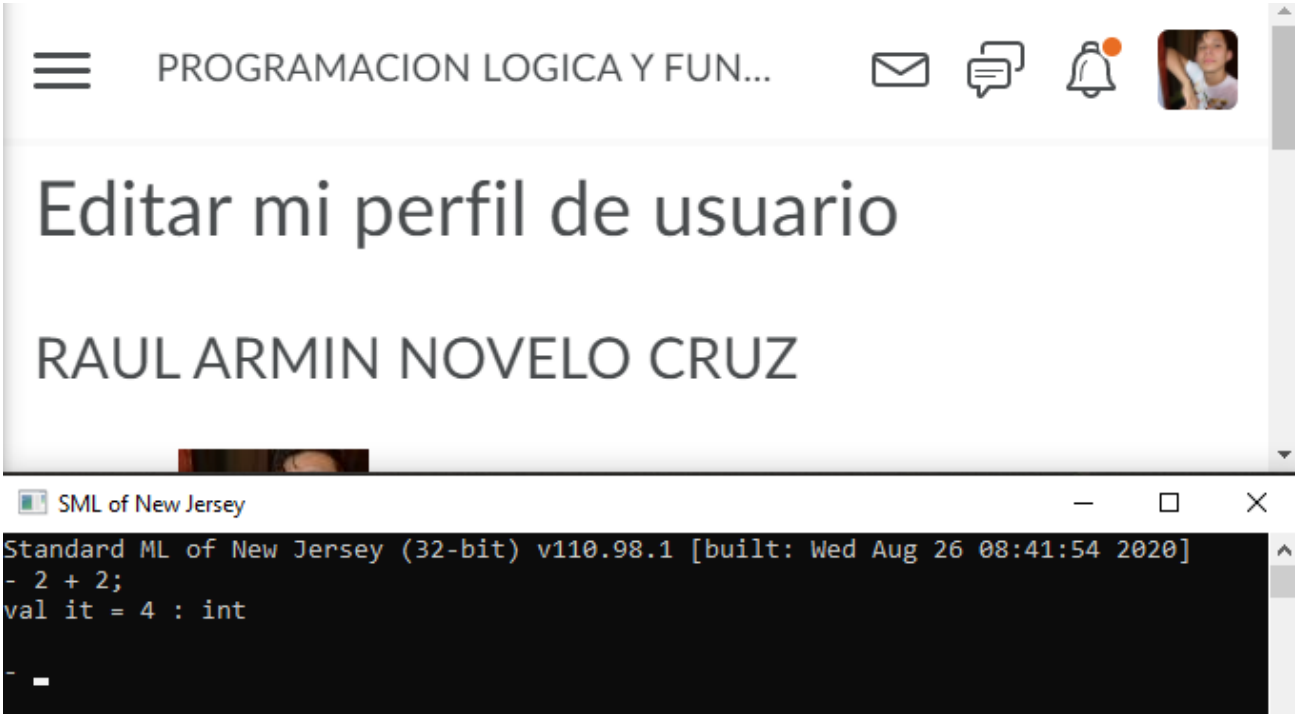
Desde el REPL, podemos escribir expresiones para que SML/NJ las evalúe. Por ejemplo, si queremos agregar
2 + 2, escribimos 2 + 2;

**Tarea 1.**

Tipo

2 + 2;

en el REPL y pulse Intro. ¿Cuál es la salida?




La salida debería haber sido

val it 4 : int

Esto indica el tipo y el valor de la evaluación de la expresión, así como el nombre al que se enlazaba este valor. se utiliza como nombre predeterminado para el valor si un nombre no es proporcionado por usted, 4 es el valor o resultado, y int es el tipo de la expresión - un entero.

SML utiliza tipos para garantizar en tiempo de compilación que los programas no pueden salir mal de ciertas maneras; la frase 4 : int se puede leer como "4 tiene el tipo int".

Observe que la expresión se terminó con un punto y coma; si no lo hacemos, el REPL no sabe evaluar la expresión y espera más entrada.

Tenga en cuenta que los punto y coma *no* son necesarios al escribir código SML en un archivo. Además, cualquier código que envíes para la tarea no debe usar punto y coma, o puedes perder puntos de estilo.[1]

[1] Con la rara excepción de Uso Declaraciones.

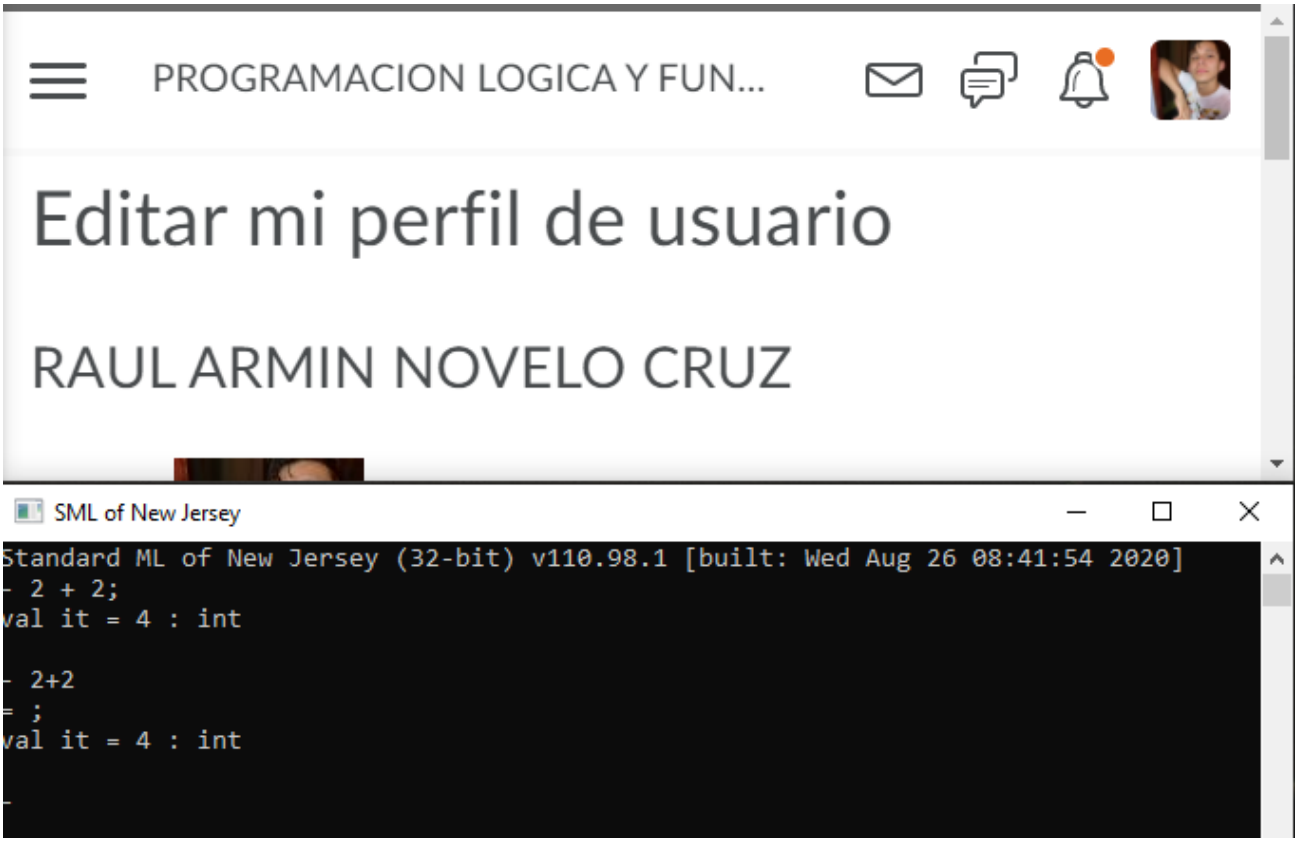
**Tarea 2.**

Tipo

2 + 2

en el REPL y pulse Intro. ¿Cuál es la salida?

Después de hacer eso, ingrese un punto y coma. ¿Qué pasa ahora?




Como puede ver, es posible poner el punto y coma en la siguiente línea y seguir obteniendo el mismo resultado.

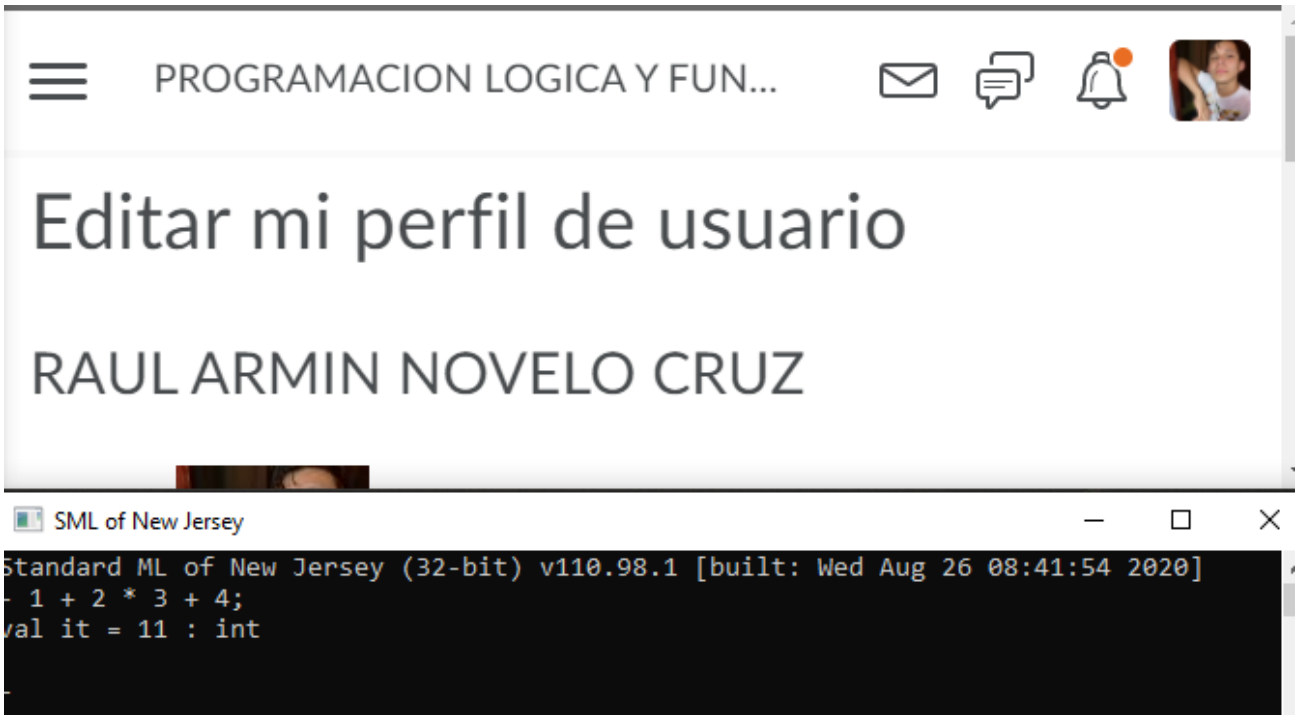
## 3.1 Paréntesis

En una clase aritmética hace mucho tiempo, probablemente aprendió algunas reglas estándar de prioridad del operador, por ejemplo, multiplicar antes de agregar, pero cualquier cosa agrupada entre paréntesis se evalúa primero. SML sigue exactamente las mismas reglas de prioridad (PEMDAS!!). Puede insertar paréntesis en expresiones para forzar un orden de evaluación determinado.

**Tarea 3.**

Tipo

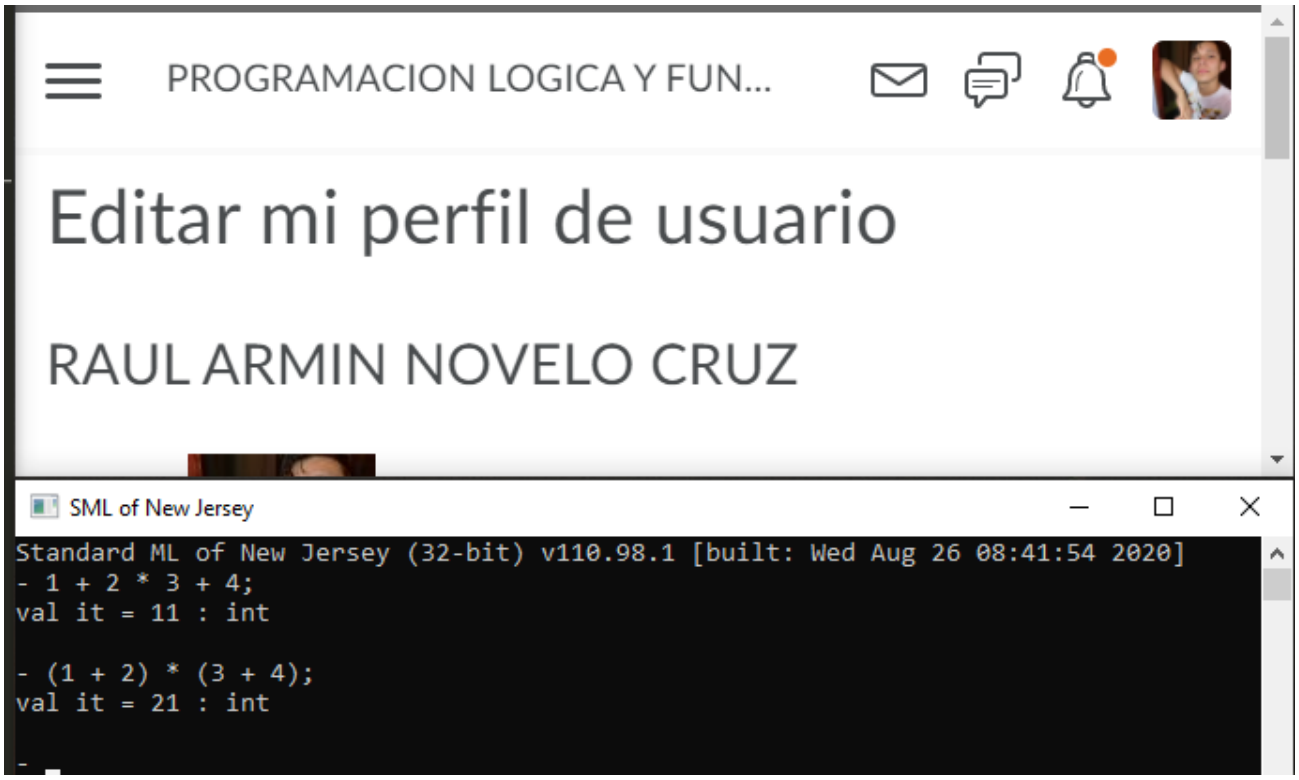
1 + 2 * 3 + 4;




en el REPL. ¿Cuál esperaría que sea el resultado? **11** ¿Cuál es el resultado real? **21**

Ahora, escriba

(1 + 2) * (3 + 4);




en el REPL. ¿El resultado es el mismo? **NO** ¿Por qué o por qué no?? **Por la precedencia de los paréntesis**

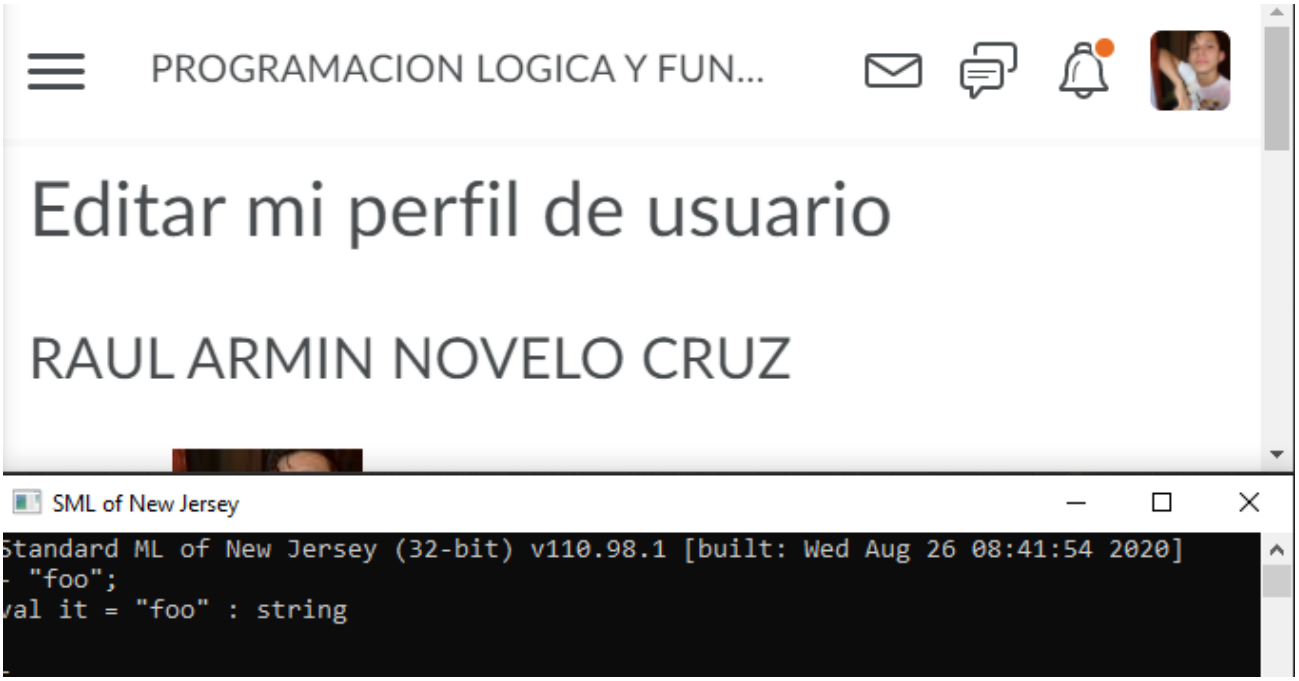
# 4 Tipos

Hay muchos tipos en SML; más que sólo int! Por ejemplo, hay una cadena de tipo para cadenas de texto.

**Tarea 4.**

Escriba "foo"; en el REPL. ¿Cuál es el

resultado?



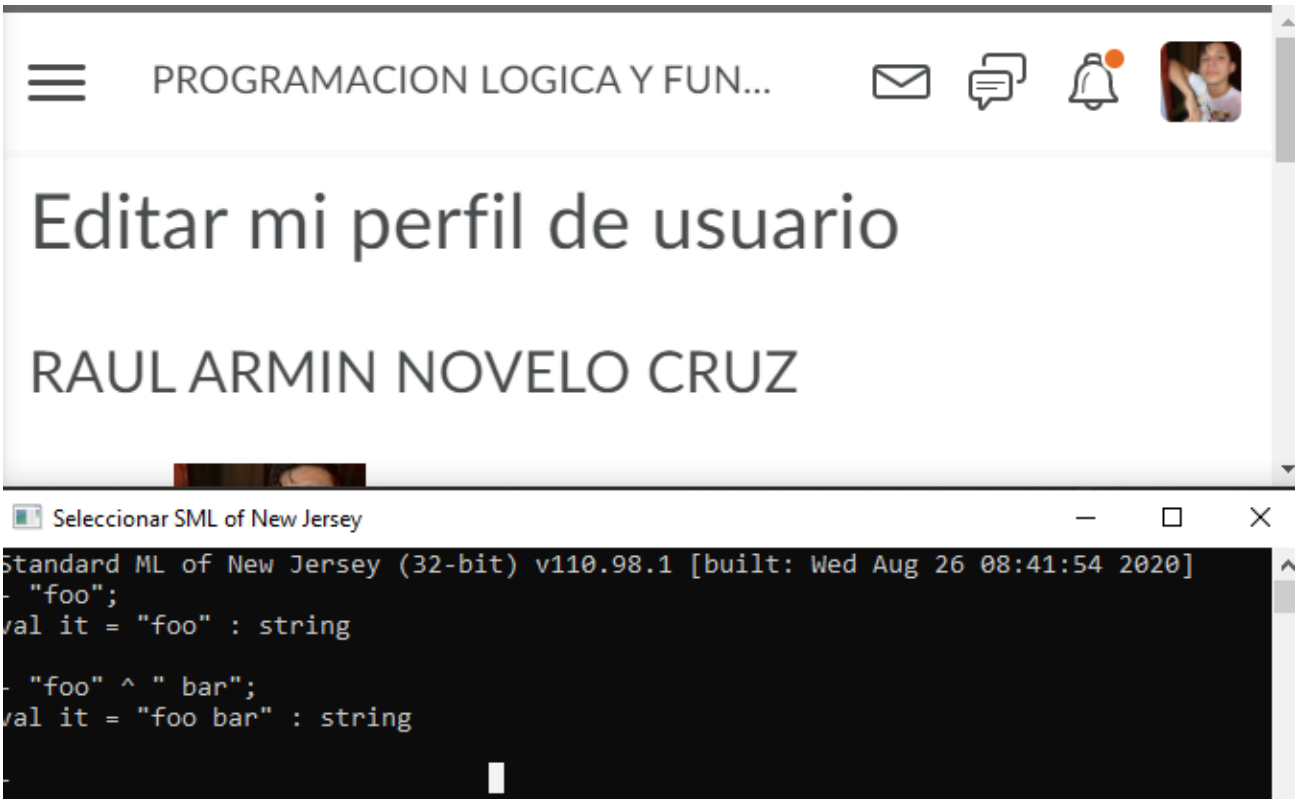

En lugar de ver un número como la salida, verá una cadena aquí. Es posible concatenar dos cadenas, utilizando el operador infix . Infix significa insertar la función entre sus argumentos, por lo que esto se utiliza en cadenas similares a cómo se utiliza + en enteros.

**Tarea 5.**

Tipo

"foo" á " bar";

en el REPL. ¿Cuál es el resultado?



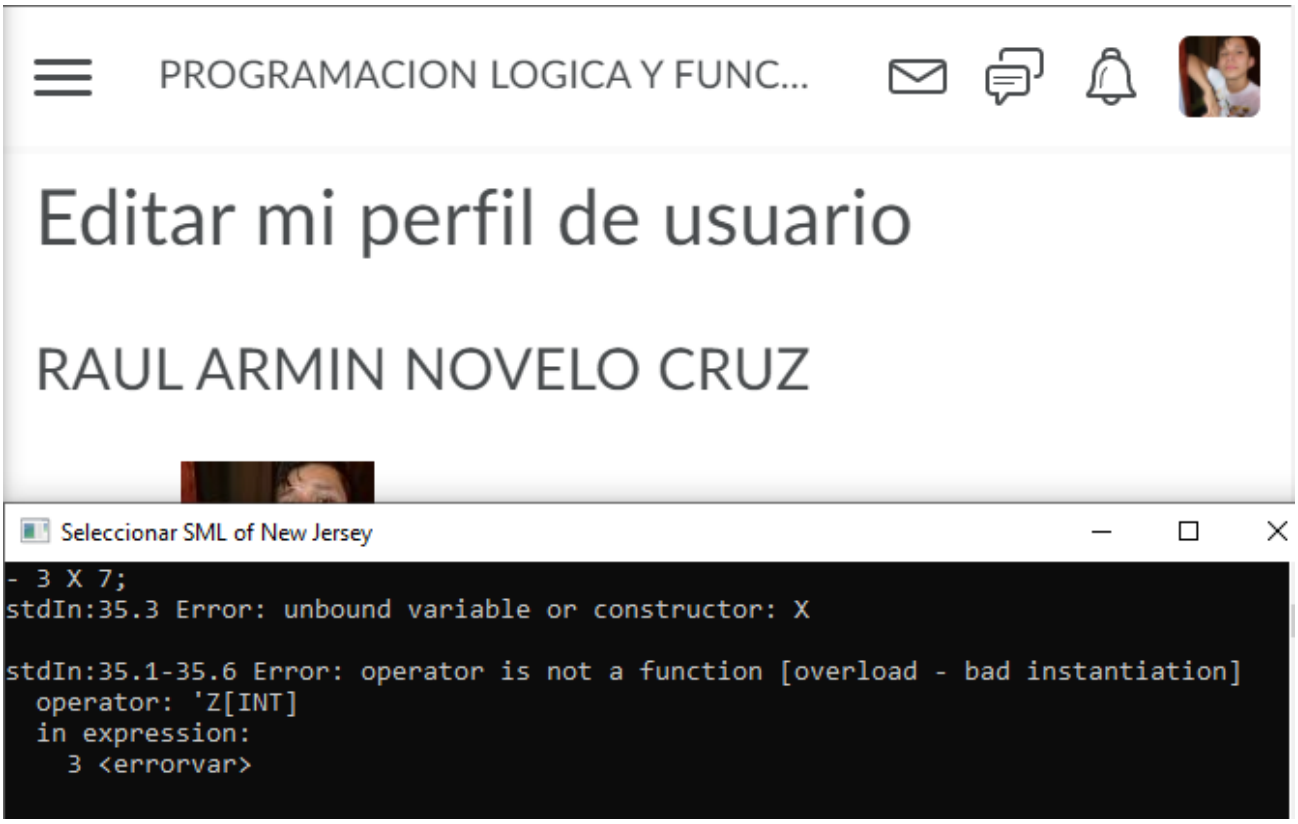

Podemos escribir un programa que no escribacheck (tiene una expresión que no está bien tipada) para ver lo que SML hace en esa situación. Por ejemplo, la concatenación solo funciona en dos cadenas.

**Tarea 6.**

Qué sucede cuando escribes

3 x 7;

en el REPL? Ocurre un error ya que SML intenta interpretar la expresión como una función y x como algún parámetro recibido/declarado.




Este es un ejemplo de uno de los mensajes de error de SML - usted debe comenzar a familiarizarse con ellos, ya que los verá bastante este semestre, al menos hasta que se acostumbre a los tipos!

# 5 Variables

Arriba, mencionamos que los resultados de los cálculos están enlazados a la variable de forma predeterminada. Esto significa que una vez que hemos hecho un cálculo, podemos referirnos a su resultado en el siguiente cálculo.

**Tarea 7.**

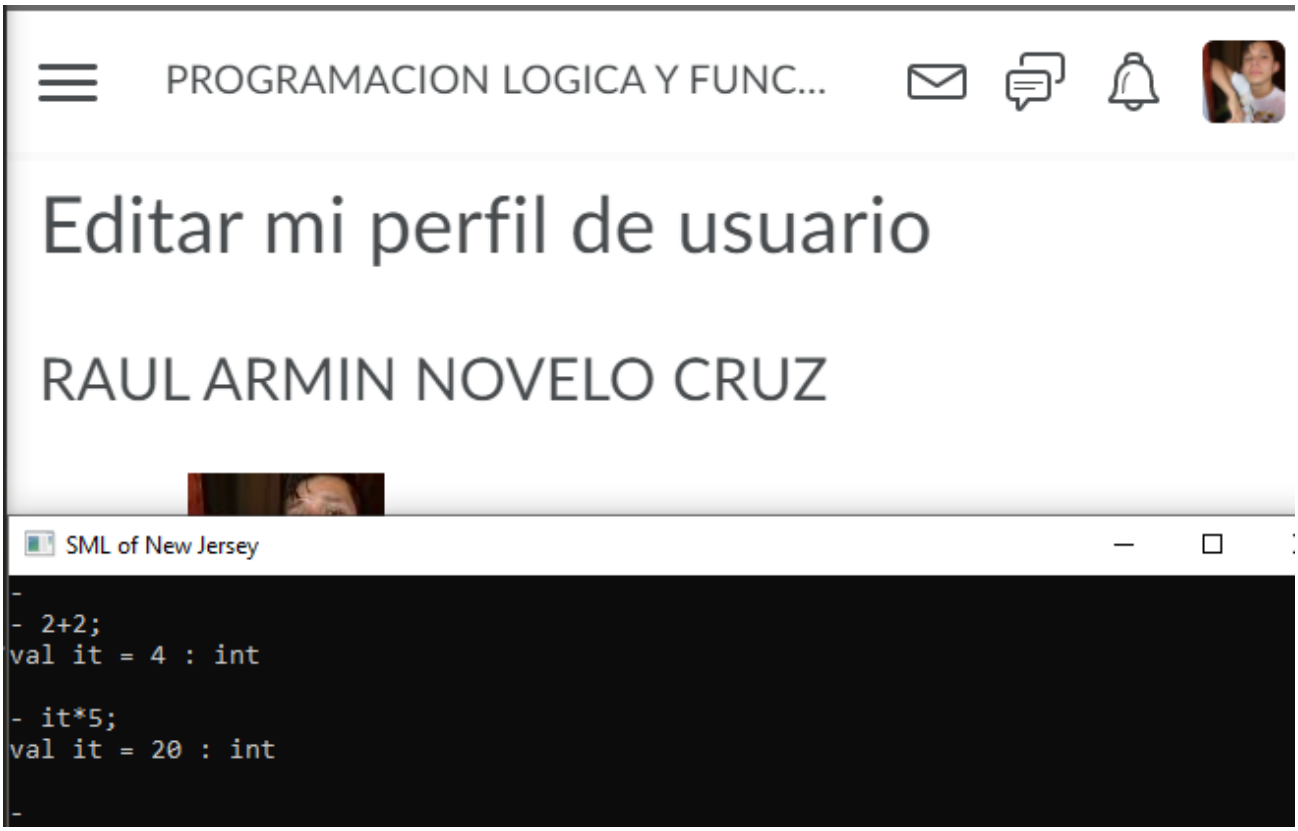
Tipo

2 + 2;

en el REPL. A continuación, escriba

es * 5;

en el REPL. ¿Cuál es el resultado?




Como puede ver, antes de la segunda evaluación, el valor enlazado a él era 4 (el valor de 2 + 2), y ahora está enlazado a 20, el resultado de la evaluación de expresión más reciente (el valor de la misma * 5 con él enlazado a 4).

Por supuesto, usted no debe entrar en el hábito de usarlo así! El sistema de tiempo de ejecución SML solo lo utiliza (es decir, él)como un valor predeterminado conveniente y una forma de ayudarle a depurar código. La sintaxis SML para la declaración de valor le permite *enlazar* un identificador (o variable) a un valor, lo que le permite hacer referencia a él con un nombre más mnemotécnico.

Una declaración de valor utiliza la palabra clave val y tiene la sintaxis:

val <varname> ? <expr>

Esta declaración intenta evaluar la expresión <expr> a un *valor* y, a continuación, *enlaza* ese valor a la *variable* (o identificador) denominada <varname>.[2]

Si queremos indicar explícitamente el tipo deseado de la variable, podemos darle una *anotación de tipo* de la siguiente manera:

val <varname> : <tipo> ? <expr>

Preferimos escribir anotaciones como se hizo anteriormente, pero también podemos escribir:

val <varname> ?<expr> : <tipo>

[2] A veces llamamos a esto *la creación de un Val Vinculante*. Los enlaces se pueden combinar para formar declaraciones. Un enlace es una declaración atómica. Cada uso de Val es un enlace, y una sola declaración puede consistir en muchos. Dos declaraciones se pueden combinar simplementeng ellos uno tras otro, que podríamos considerar como una sola declaración.

O incluso:

val <varname> : <tipo> ?<expr> : <tipo>

Las declaraciones tienen un atributo denominado *scope*. El ámbito de una declaración está donde se puede usar. [3]Si podemos declarar algo "cosa" utilizando una declaración, entonces decimos que "cosa" está *dentro del ámbito* de esa declaración.

Podríamos hacer declaraciones en el REPL SML/NJ, pero también podemos hacerlas como parte de una expresión letusando la sintaxis:

let val <varname> ? <expr1> en <expr2> end

El valor de estalet -expression se obtiene evaluando <expr1>, enlazando su resultado a la variable denominada <varname>, y utilizando esta declaración al evaluar <expr2>. Esta declaración no está disponible para su uso fuera de <expr2>, y decimos que el *ámbito* de esa declaración es esta expresión.

La sintaxis de declaración SML es mucho más general de lo que hemos indicado aquí, pero esto introduce las nociones clave de ámbito y enlace.

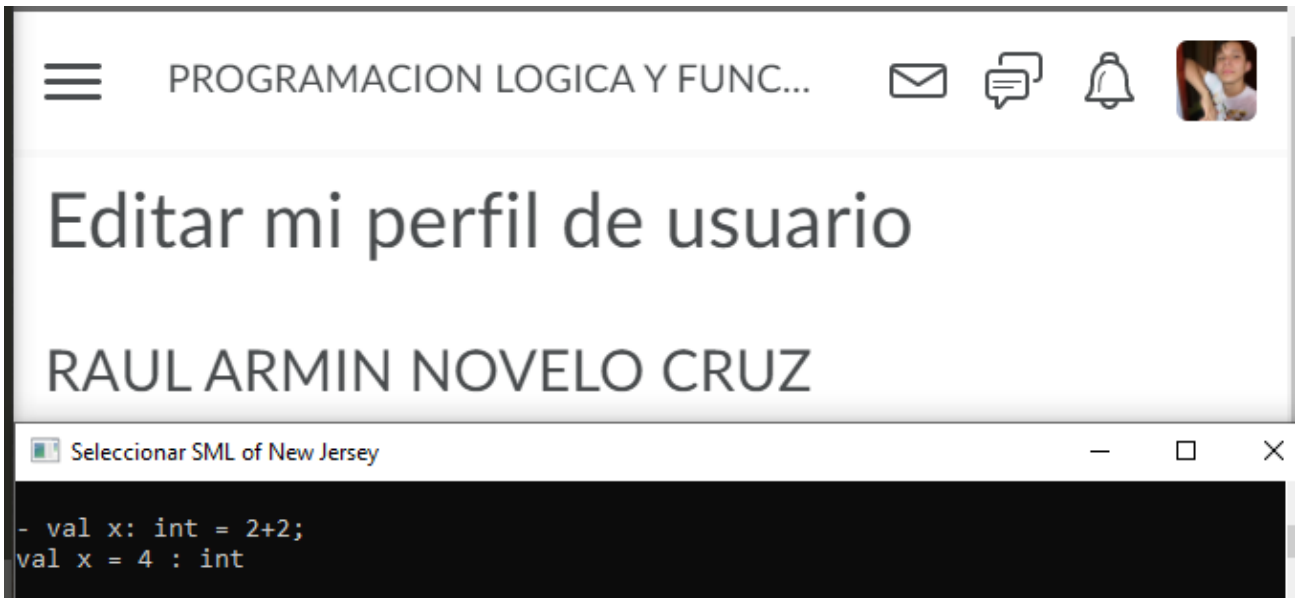
**Tarea 8.**

Tipo

val x : int á 2 + 2;

en el REPL. ¿Cuál es el resultado? ¿En qué se diferencia de simplemente escribir lo siguiente?

2 + 2;

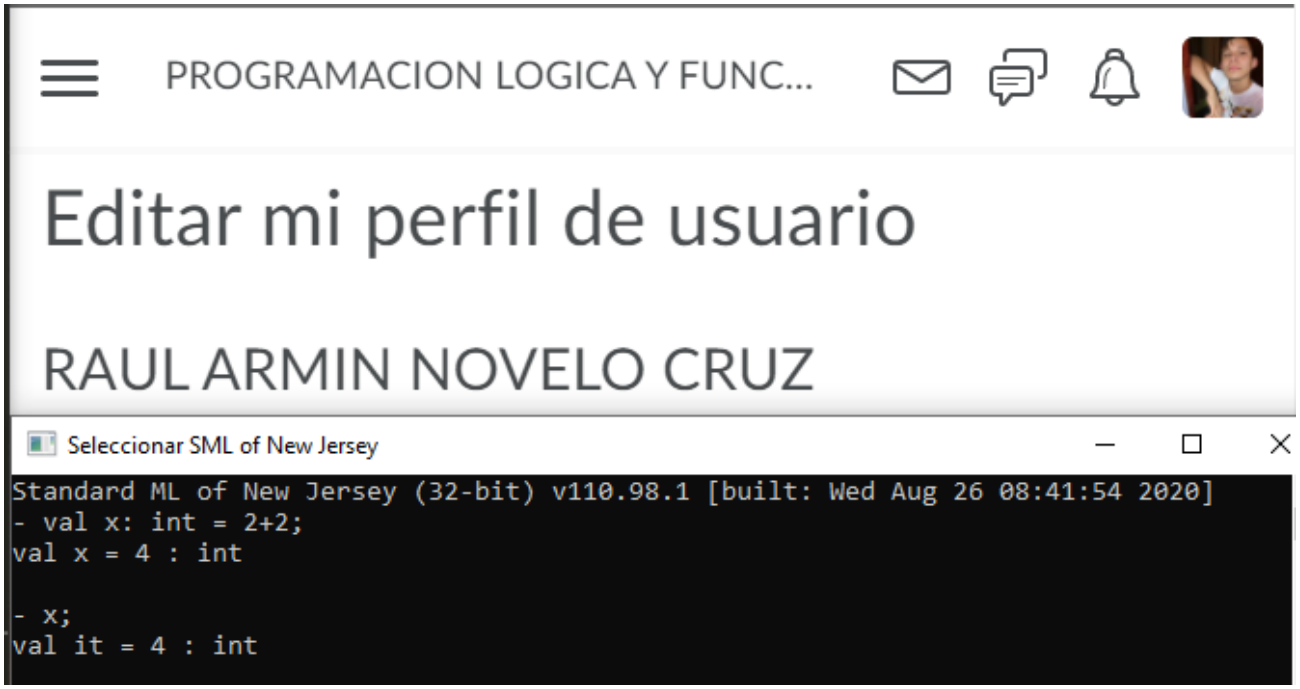



Como puede ver, esa declaración enlaza el valor de 2 + 2 a la variable x. Ahora podemos usar x.

[3] El concepto de alcance se utilizará durante todo el semestre; si usted tiene alguna pregunta, por favor pregunte a sus TAs!

**Tarea 9.**

Tipo x; en el REPL. ¿Cuál es el resultado?

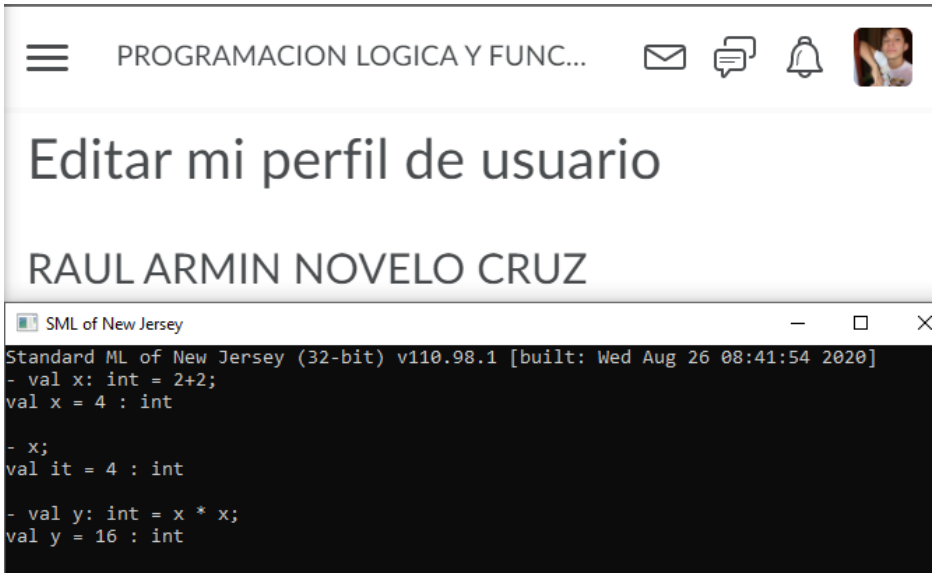



**Tarea 10.**

Ahora escriba

val y : int á x * x;

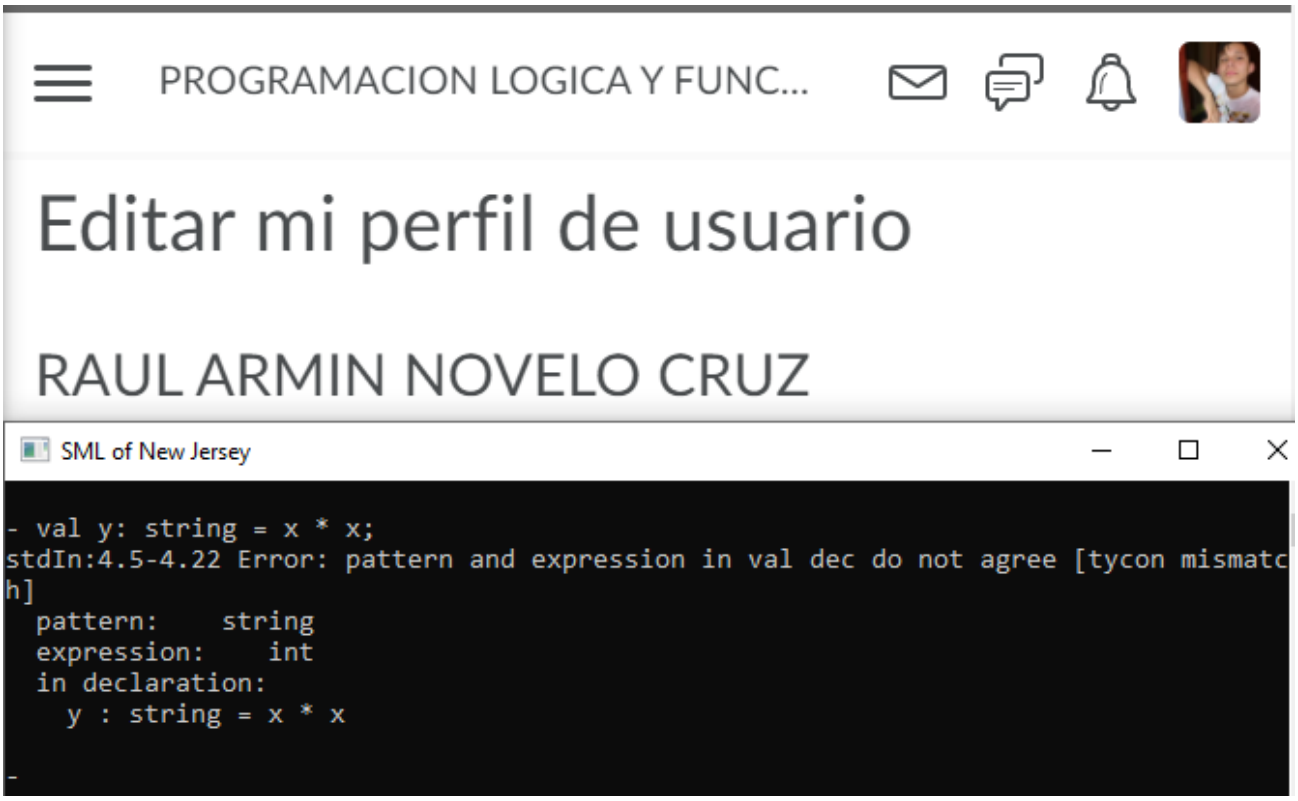
en el REPL. ¿Cuál es el resultado?



**Tarea 11.**

¿Qué hay de

val y : string á x * x;



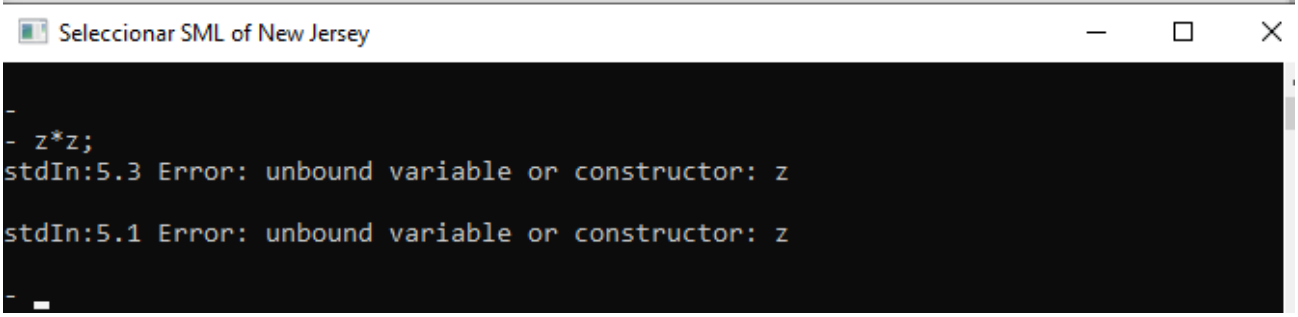

¿Qué pasa? ¿por qué?

**Ocurre un error debido a que no es posible convertir a string el resultado numérico**

**Tarea 12.**

Después de eso, escriba

z * z;


en el REPL. ¿por qué?

**R: Arroja un erro debido a que z no ha sido declarado como variable y se desconoce su tipo**

Las variables de SML hacen referencia a valores, pero no se *pueden reasignar* como variables en lenguajes de programación imperativos.

Cada vez que se declara una variable, SML crea un nuevo *enlace* y enlaza esa variable a un valor. Este enlace está disponible, sin cambios, en todo el *ámbito* de la declaración que lo introdujo.

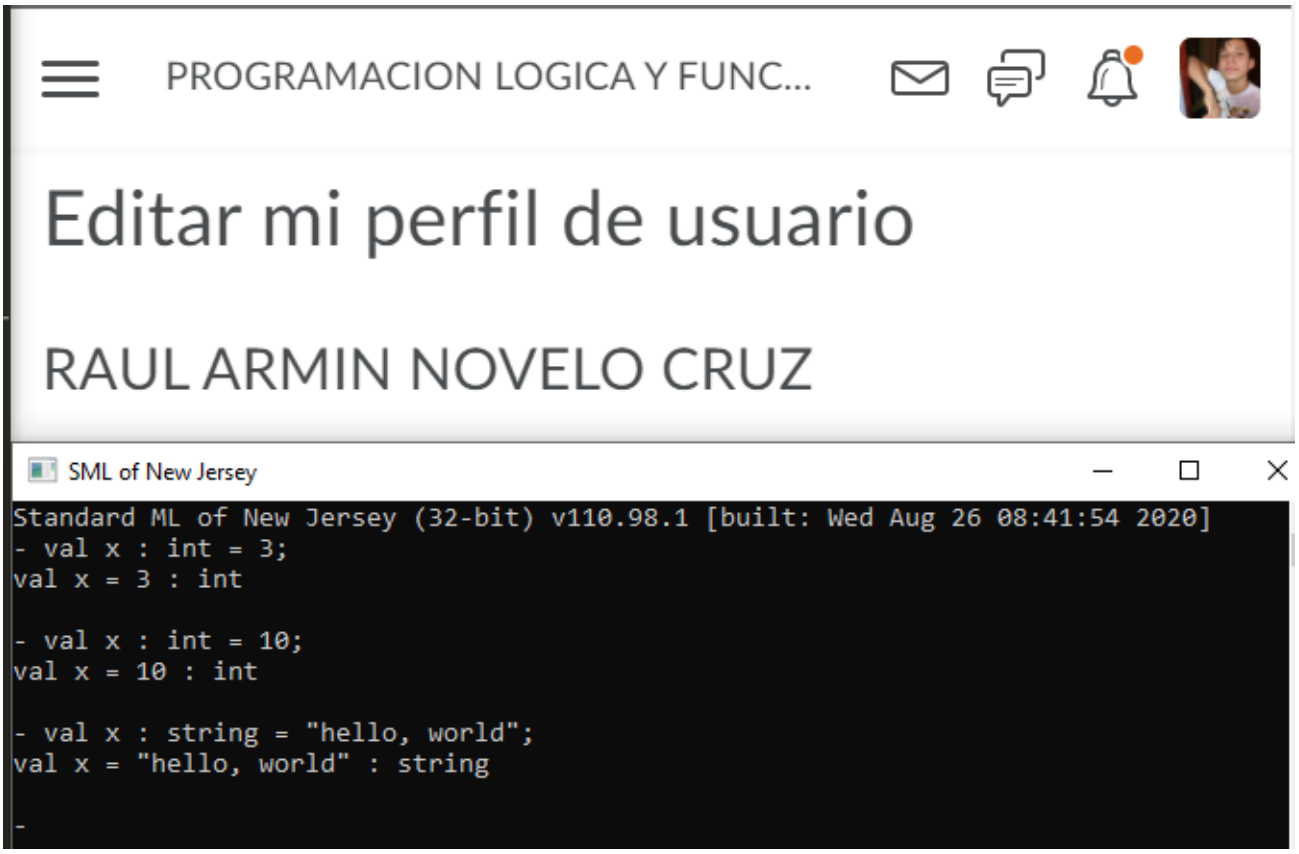
Si el nombre se usó antes, el nuevo enlace *sombrea* el anterior: el enlace antiguo todavía está alrededor, pero los nuevos usos de la variable hacen referencia al más reciente.

**Tarea 13.**

Tipo

val x : int á 3; val x : int á 10; val x : string á "hola, mundo";

```
val x : int = 3;
val x : int = 10;
val x : string = "hello, world";
```




en el REPL. ¿Cuáles son el valor y el tipo de x después de cada línea?

**3 e int, 10 e int , "hello, world" y string, respectivamente**

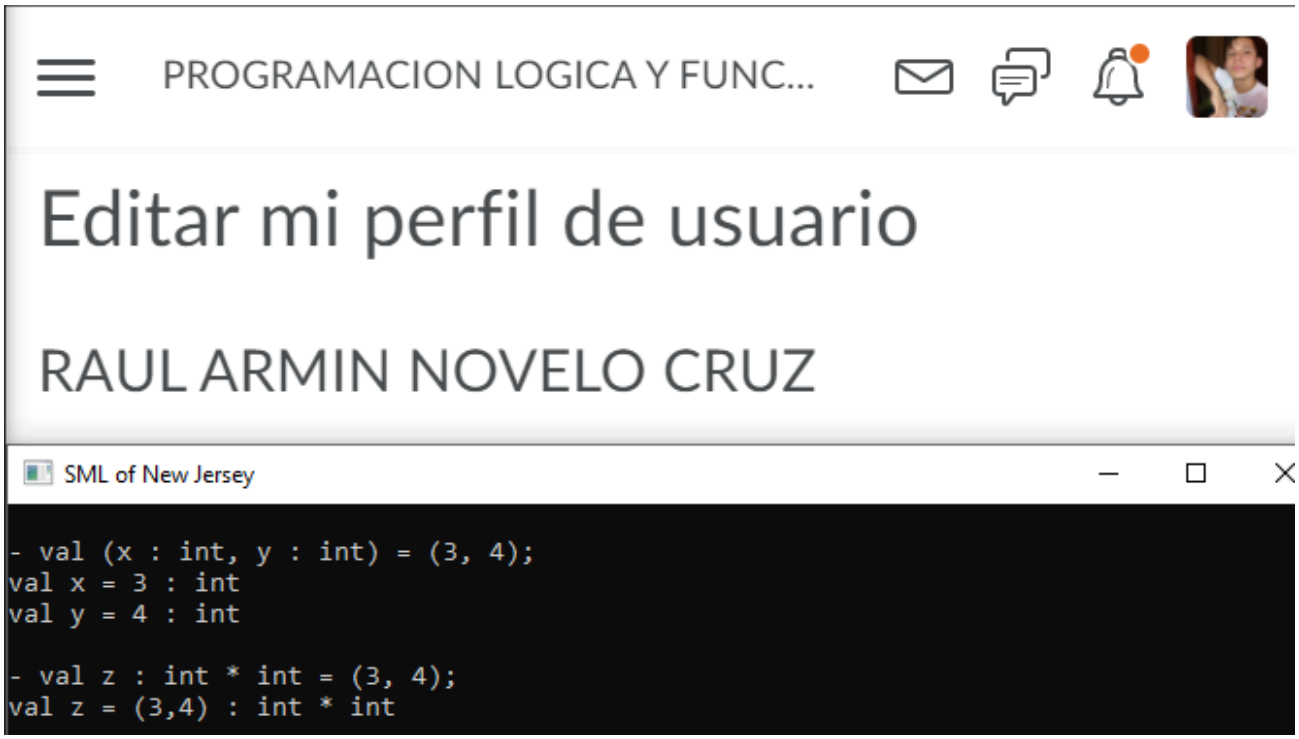
Podemos introducir varios identificadores en una sola declaración mediante tuplas. Cuando escribimos una tupla, sus elementos están separados por comas:

val (x : int,y : int)á (3,4)

Esta declaración enlaza x a 3 e y a 4.

*Una tupla de valores es en sí misma un*valor, por lo que también podemos hacer esto:

val z : int * int á (3,4)




Esto se une a z a la tupla (3, 4). Tenga en cuenta que las tuplas tienen lo que se llama un tipo de *producto,*indicado por el * en él. Una tupla tendrá * en su tipo para separar los tipos de sus elementos. Para 3 tuplas, habría dos *'s en su tipo.

¡Las declaraciones son útiles de otras maneras! Podemos usarlos para escribir pruebas para funciones. Supongamos que queríamos probar los operadores integrados + y . Podríamos escribir:

```
val 3 : int á 1 + 2; val 10 : int á
5 + 5;
val "hola, mundo" - "hola", "mundo";;
```

SML/NJ no generará nada porque no creamos nuevos enlaces; 3 siempre enlaza 3, "hola, mundo" siempre une "hola, mundo",etc.

Los valores *siempre* se unen a sí mismos, y usaremos este hecho para escribir algunas pruebas para otras funciones
más tarde.[4]

# 6 Funciones

## 6.1 Uso de archivos

Hemos escrito algunas expresiones SML básicas, por lo que ahora podemos echar un vistazo a la obtención de entrada de archivos. Hemos proporcionado el código de archivo/práctica/playground.sml en el archivo tar que descargó.

En primer lugar, muévase al directorio code/practice/: directory:

```
> código ls
README.md
> cd code/practice/
```

[4] El único *Valor* que un valor puede enlazar es *Sí mismo.*

```
> ls
playground.sml
```

A continuación, desde el REPL SML/NJ, escriba `use "playground.sml";` . La salida de SML debe ser similar a

```
- utilizar "playground.sml";
[apertura playground.sml] ...
Val it () : unidad
-
```

Ahora que ha hecho esto, tiene acceso a todo lo que se definió en code/practice/playground.sml, como si hubiera copiado y pegado el contenido del archivo en la REPL.

Si esto no funciona, es posible que no esté en el directorio correcto. Salga de la REPL presionando Ctrl+D, luego escriba `pwd` y `ls` para comprobar que está en el directorio correcto y el archivo playground.sml existe en el directorio de trabajo.

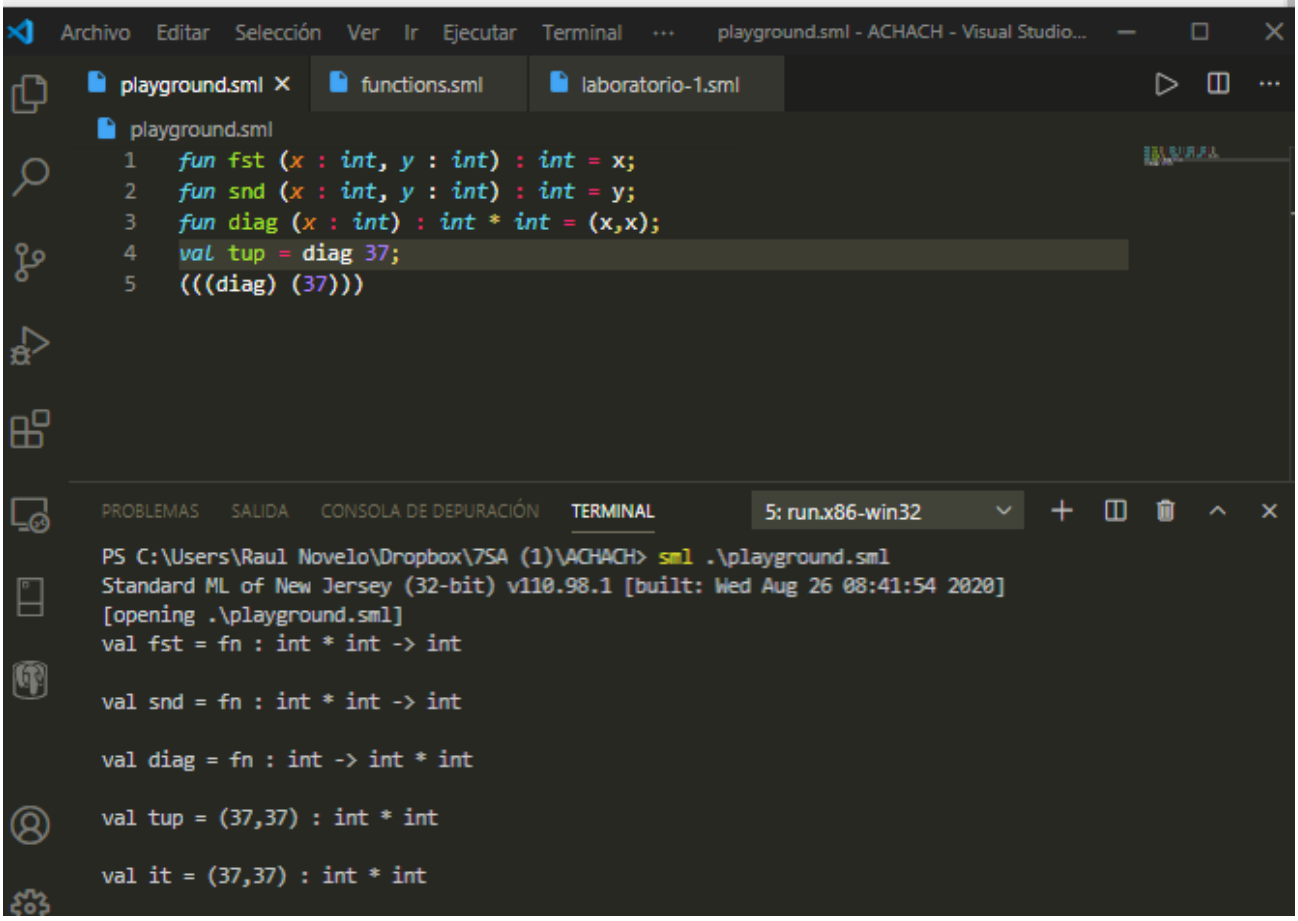
## 6.2 Aplicación de funciones

En este archivo, observe que hay funciones definidas. Por ejemplo, hay

```
fun fst (x : int, y : int) : int = x;
fun snd (x : int, y : int) : int = y;
fun diag (x : int) : int * int = (x,x);
```

Observe que el tipo de valor devuelto de diag es `int * int,` lo que significa que devuelve una tupla de ints, por ejemplo
(1,1) . La función diag se puede invocar escribiendo `diag(37)`. Sin embargo, los paréntesis alrededor del argumento son realmente innecesarios. No importa si escribimos `diag 37` o `(((diag) (37)))` – ambos se evalúan exactamente igual.

RAUL ARMIN NOVELO CRUZ




Buen estilo SML utiliza sólo tantos paréntesis como sea *necesario* para dejar clara su intención. Por ejemplo, aunque

2 * 6 + 3 * 5 (2 * 6) + (3 * 5)

evaluar a la misma cosa, preferimos la segunda forma, porque es más claro ver el orden de las operaciones.

Sin embargo, dado que

diag 5

se analiza de la misma manera que

diag(5)

mientras que también usamos menos caracteres, preferimos la primera forma.

Tenga en cuenta que las dos expresiones siguientes son muy diferentes, por razones que exploraremos más adelante en el curso.

fst diag(5) fst (diag 5)

Esta es otra razón para preferir el uso mínimo de paréntesis durante la aplicación de la función.

**Tarea 14.**

¿Cuál es el tipo de fst, y qué hace?

**R: es de tipo int y recibe 2 parámetros (x y) y retorna x**

**Tarea 15.**

¿Cuál es el tipo de sndy qué hace?

**- R: es de tipo int y recibe 2 parámetros (x, y) y retorna y**

**Tarea 16.**

¿Cuál es el tipo de diag y qué hace?

**R: es de tipo int y recibe 1 parámetro x, y retorna una tupla de x: (x, x)**

## 6.3 Definición de funciones

Por lo general, se requiere que utilice un formato estándar para comentar definiciones de función: el código SML para una definición de función debe ir precedido de un comentario que especifique el tipo de la función, una cláusula "requiere" y una cláusula "asegura". Llamamos a esto *la especificación de una función*. Una cláusula REQUIRES es una garantía lógica sobre la entrada (es decir, suposiciones que está haciendo sobre la entrada), mientras que una cláusula ENSURES le indica lo que está garantizando sobre la salida. Estos nos permiten razonar formalmente sobre el comportamiento de las funciones, que es una parte clave de 15-150.

Podemos renunciar a este requisito cuando la función forma parte de la biblioteca de bases de SML o se ha especificado anteriormente. Cuando la función no tiene condiciones previas, utilizamos la notación "REQUIERE: true" (es decir, todos los argumentos del tipo correcto cumplen la condición previa). Nunca es necesario especificar en el REQUIRES que la entrada sea del tipo correcto, ni debe especificar en ensures cuál es el tipo de salida. Este es el punto de la anotación de tipo.

Por ejemplo,

```
(* incr : int -> int
*   REQUIERE: verdadero
*   ASEGURA: incr x -> el siguiente entero después de x
*)
fun incr (x : int) : int á x + 1
```

Es un poco superfluo escribir especificaciones para funciones que son tan sencillas, pero esta es la forma general que le pediremos que use al documentar su código, y será mucho más útil más adelante cuando su código se vuelva muy complejo!

Tenga en cuenta que nuestra cláusula ENSURES está escrita en inglés, no en ningún tipo de código ejecutable. Como se mencionó en la conferencia, no ejecutamos nuestras cláusulas REQUIRES y ENSURES junto con el código (como usted puede haber hecho en 15-122), sino que estas son garantías lógicas que nos ayudan, las personas que leen este código, razonar sobre él más fácilmente.

También le pedimos que pruebe su código para la corrección. Aquí hay algunas pruebas para incr:

```
val 2 á incr 1  val  5 á
incr 4 val ~3 - incr ~4
```

*Nota:*La negación numérica en SML utiliza ~, unoperador unario que tiene el tipo int -> int de forma predeterminada.
Es una función.

Lo que está sucediendo aquí es que le estamos diciendo a SML que *vincule* el valor 2 a lo que incr 1 evalúe.
Si incr 1 no es 2, entoncesSML se quejaría de que estamos tratando de enlazar 2 a algo que no es 2. Sin embargo, si incr 1 *es* 2 (como queremos), entonces sólo estamos diciendo "Hey SML, 2 se une a 2", que no tiene objeciones a.

## 6.4 Ahora es tu turno!

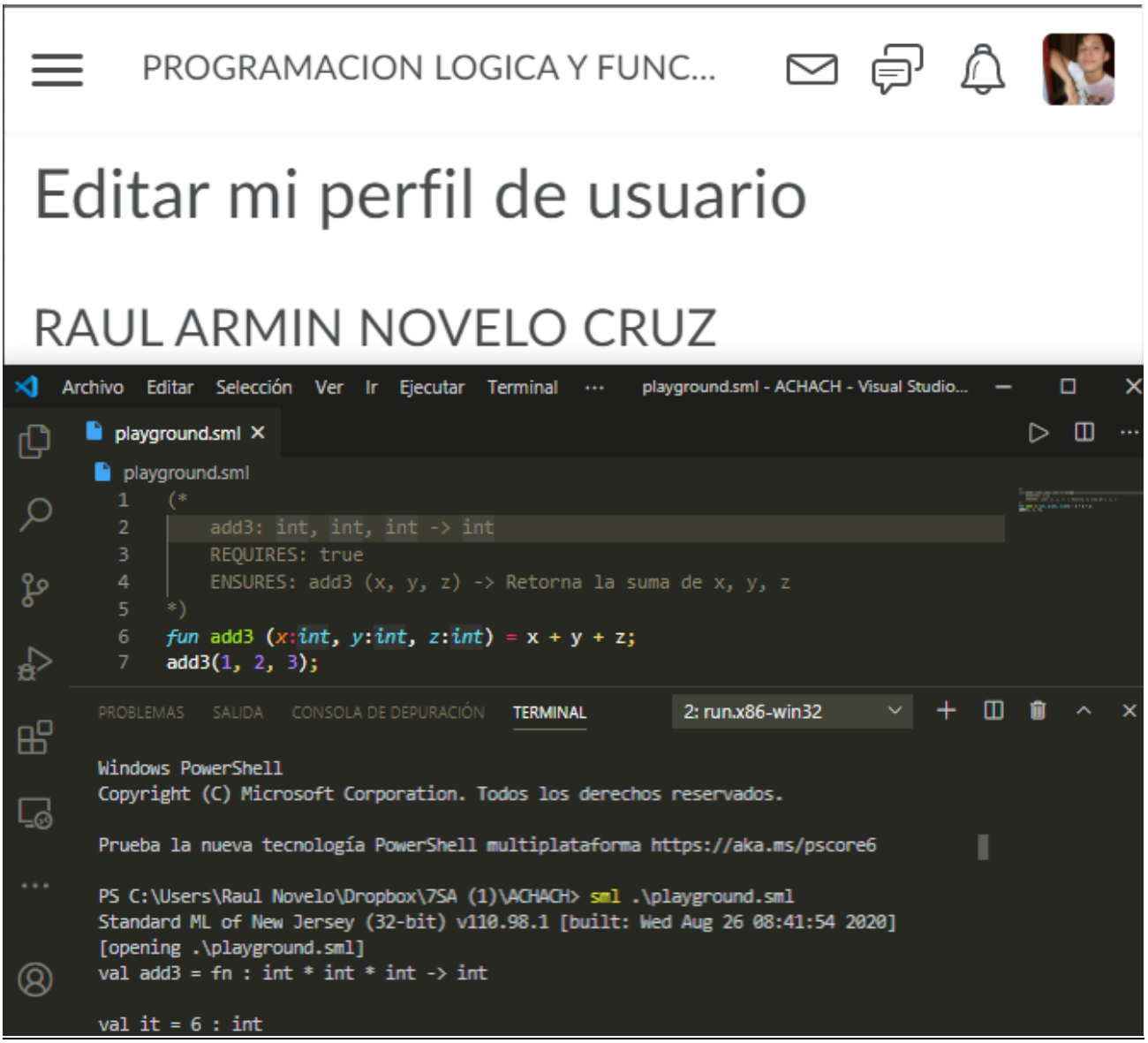
Para cada una de las siguientes funciones en code/practice/playground.sml:

1. escribir las especificaciones de la función (escribir el tipo de la función, el REQUIRES y el ENSURES)
2. implementar la función
3. escribir algunas pruebas para la función (asegúrese de que cada función pasa sus pruebas!)

**Haga todo esto en el archivo** code/practice/playground.sml; hemos incluido código de inicio para su comodidad.

**Tarea 17.**

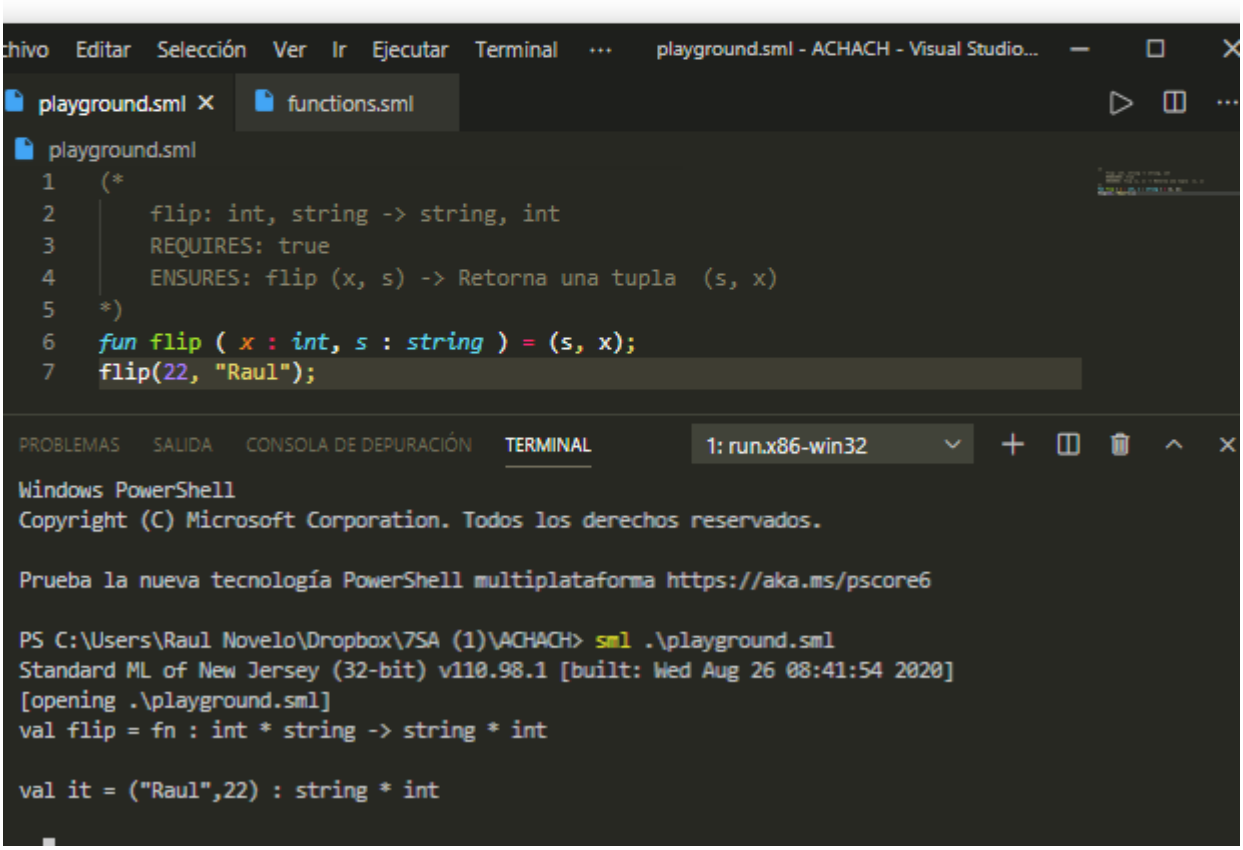
Una función denominada add3 que toma un triple (x,y,z) de tres enteros y devuelve su suma $x + y + z$.




**Tarea 18.**

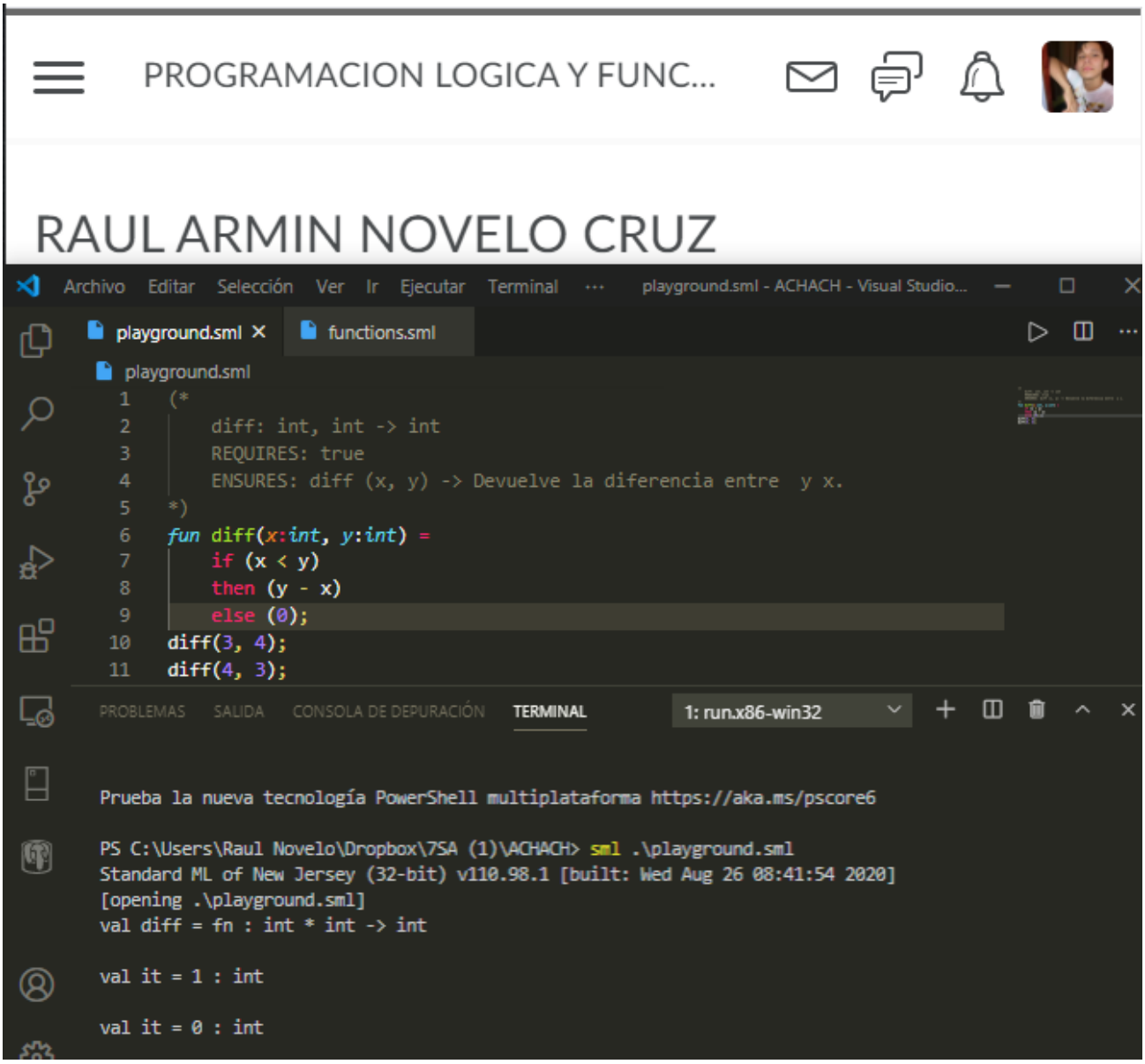
Una función denominada flip que toma una tupla (x,s) donde *x* es un entero y *s* es una cadena, y devuelve la tupla (s,x).

# ditar mi perfil de usuario




**Tarea 19.**

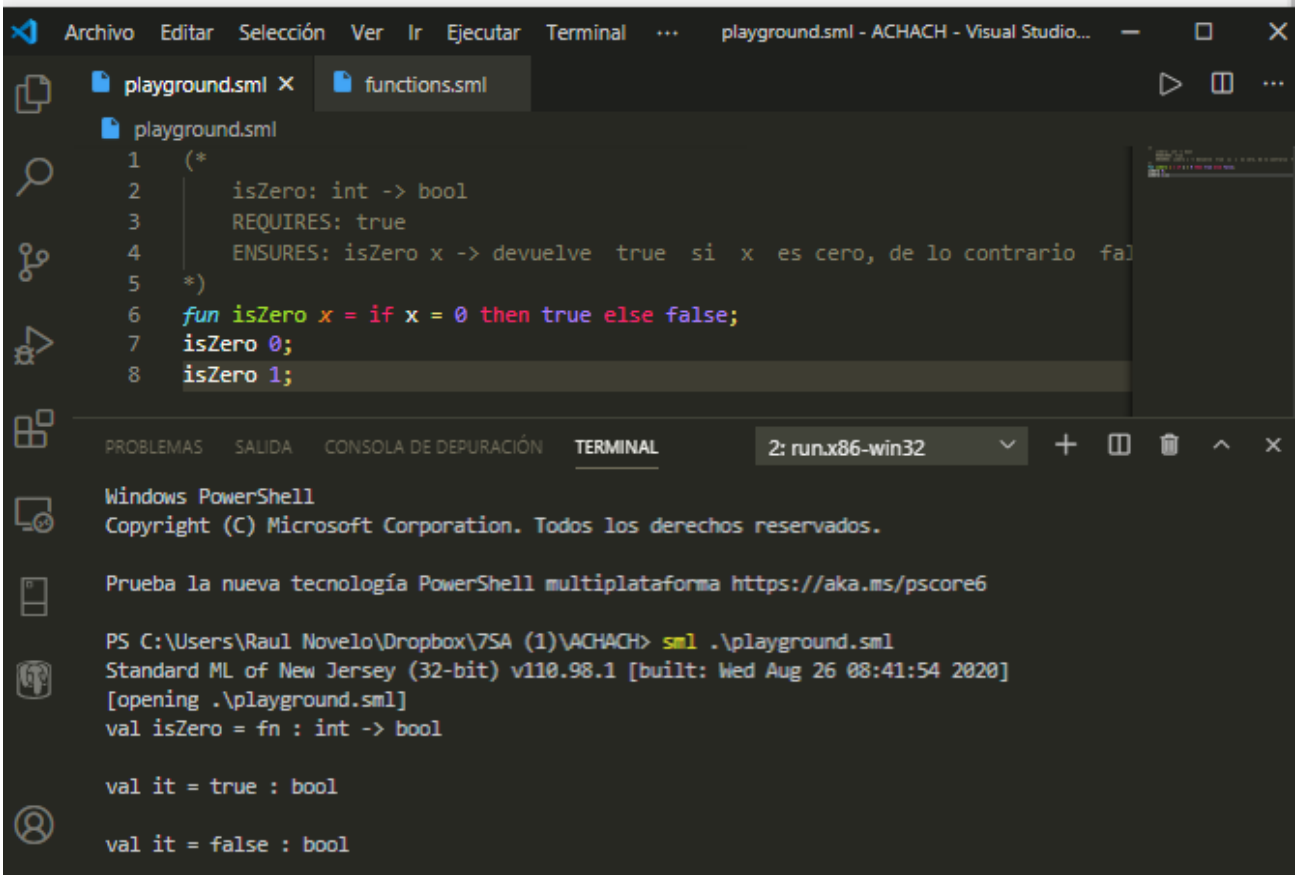
Una función denominada*x* diff que toma una tupla (x,y) de enteros donde x < *y* y devuelve su diferencia *y*x.




**Tarea 20.**

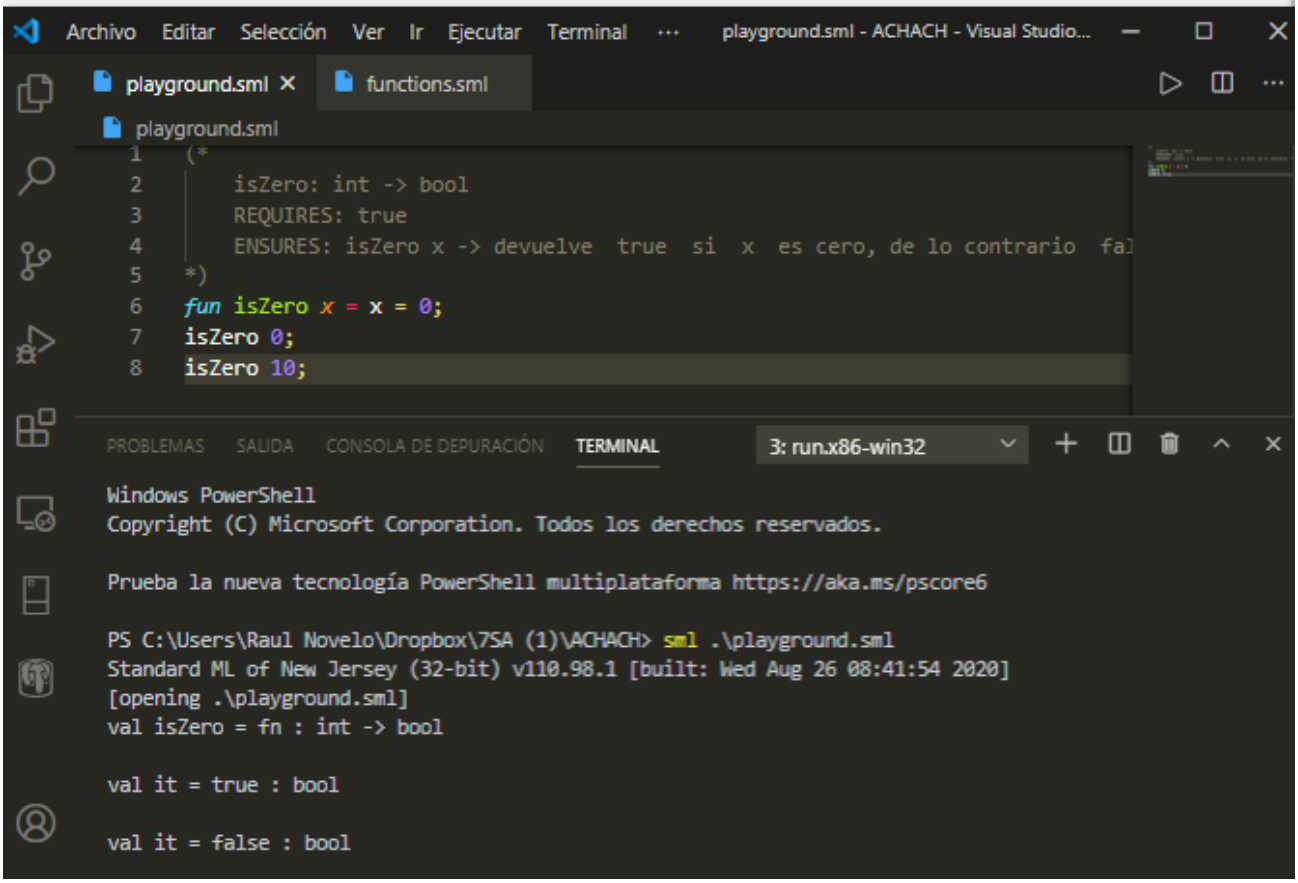
Una función denominada isZero que toma un entero x y devuelve true si x es cero, de lo contrario false.

RAUL ARMIN NOVELO CRUZ




*Nota:* Usamos el valor de la opción sml para crear enlaces, pero dentro de una expresión también es un operador. Hay dos maneras diferentes de hacer esto. ¡A ver si puedes averiguar ambas cosas!.

RAUL ARMIN NOVELO CRUZ




**Tarea 21.**

Una función denominada detectZeros que toma una tupla (x,y) y se evalúa como true si x es cero o y es cero.

*Sugerencia:* Es posible que el operador de infijo integrado os menos sea útil. Estos son algunos ejemplos de cómo se utiliza orelse: is used:

val true - true orelse true val true - true
orelse false val true - false orelse true val
false ? false orelse false

RAUL ARMIN NOVELO CRUZ