

Práctica 2

Intérprete de Lambda Cálculo

Juan Luis Filgueiras Rilo

Andrés Molares Ulloa

|

Diseño de Lenguajes de Programación – Mención en Computación – UDC

|

Curso 2018-2019

Contenido

[Ejercicio 1 1](#_Toc532558277)

[Funcionalidades Solicitadas 1](#_Toc532558278)

[Cambios Realizados 1](#_Toc532558279)

[Ejemplos de Uso 3](#_Toc532558280)

[Ejercicio 2 4](#_Toc532558281)

[Funcionalidades Solicitadas 4](#_Toc532558282)

[Cambios Realizados 4](#_Toc532558283)

[Ejemplos de Uso 7](#_Toc532558284)

[Ejercicio 3 9](#_Toc532558285)

[Funcionalidades Solicitadas 9](#_Toc532558286)

[Cambios Realizados 9](#_Toc532558287)

[Ejemplos de Uso 11](#_Toc532558288)

[Ejercicio 4 12](#_Toc532558289)

[Funcionalidades Solicitadas 12](#_Toc532558290)

[Cambios Realizados 12](#_Toc532558291)

[Ejemplos de Uso 14](#_Toc532558292)

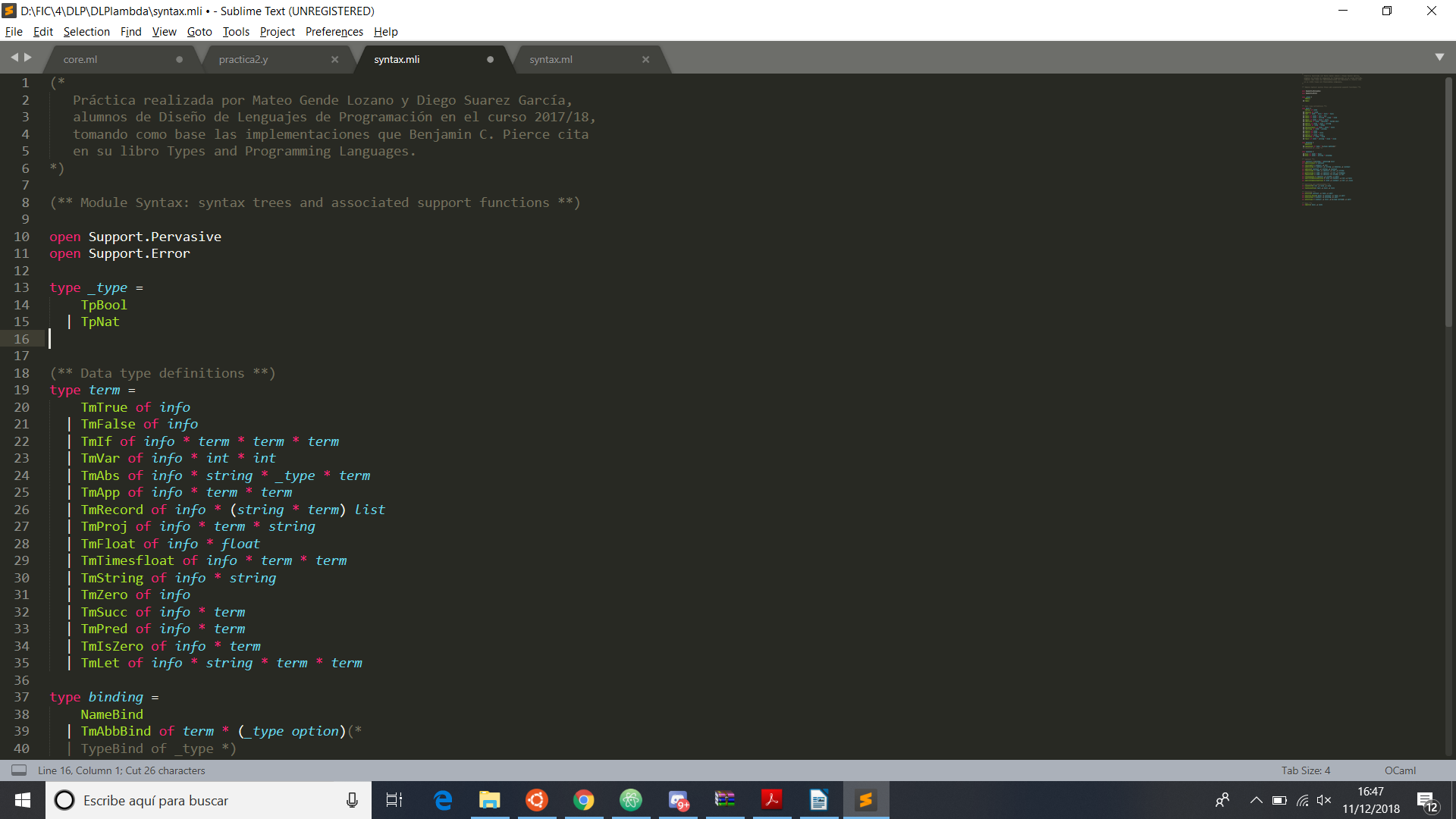
# Ejercicio 1

## Funcionalidades Solicitadas

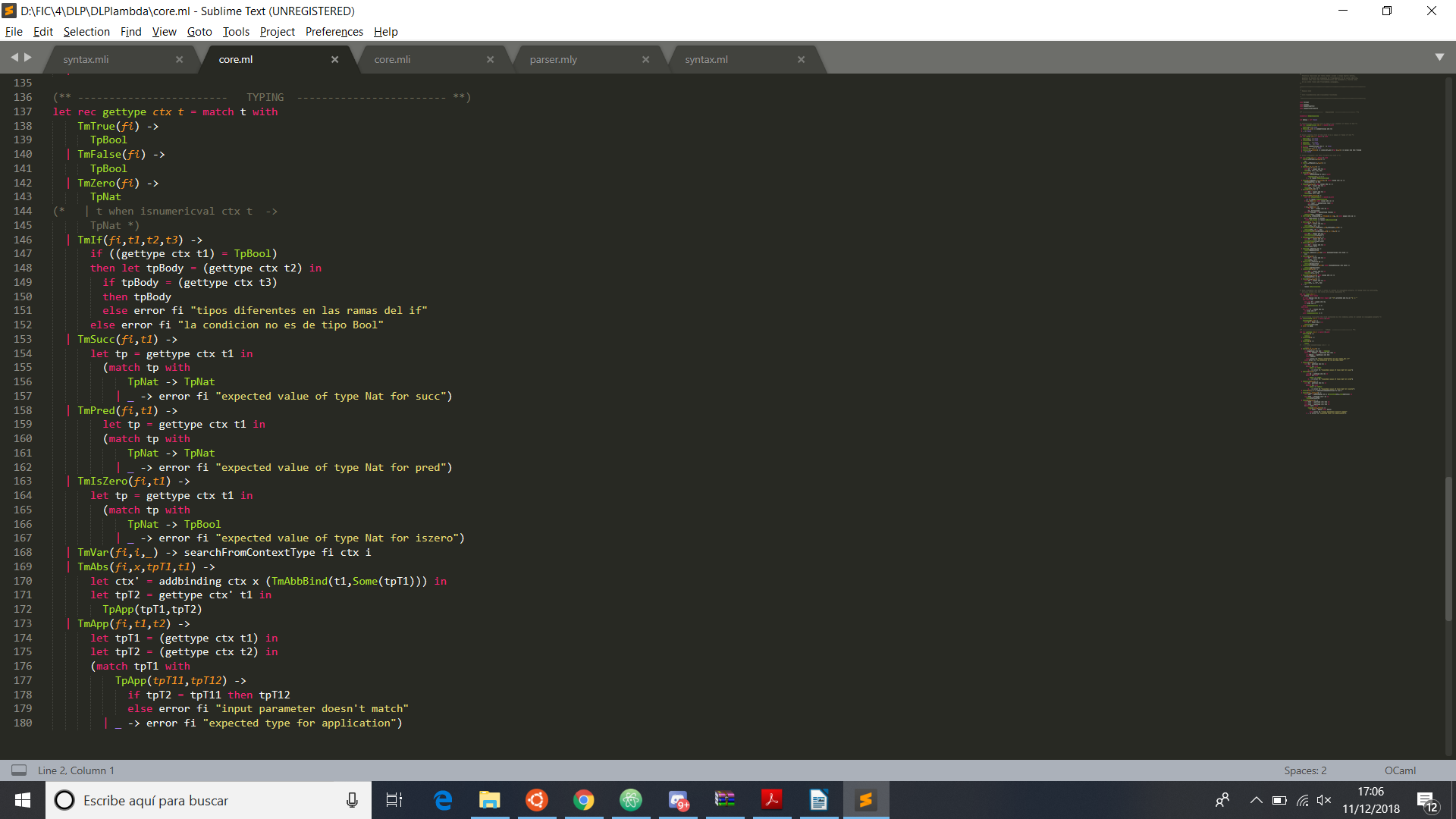
En este ejercicio se solicita una modificación del código proporcionado con el fin de detectar los errores de tipo que involucren a los tipos Nat y Bool. Para ello será necesario la incorporación al intérprete de los tipos Nat y Bool junto a los mecanismos necesarios para su correcta comprobación.

## Cambios Realizados

Para ello, introducimos en el archivo syntax.ml (y en su correspondiente interfaz de módulo, syntax.mli), a modo de definición de tipos, los tipos Nat y Bool en un nuevo tipo de datos llamado (valga la redundancia) ‘\_type’:

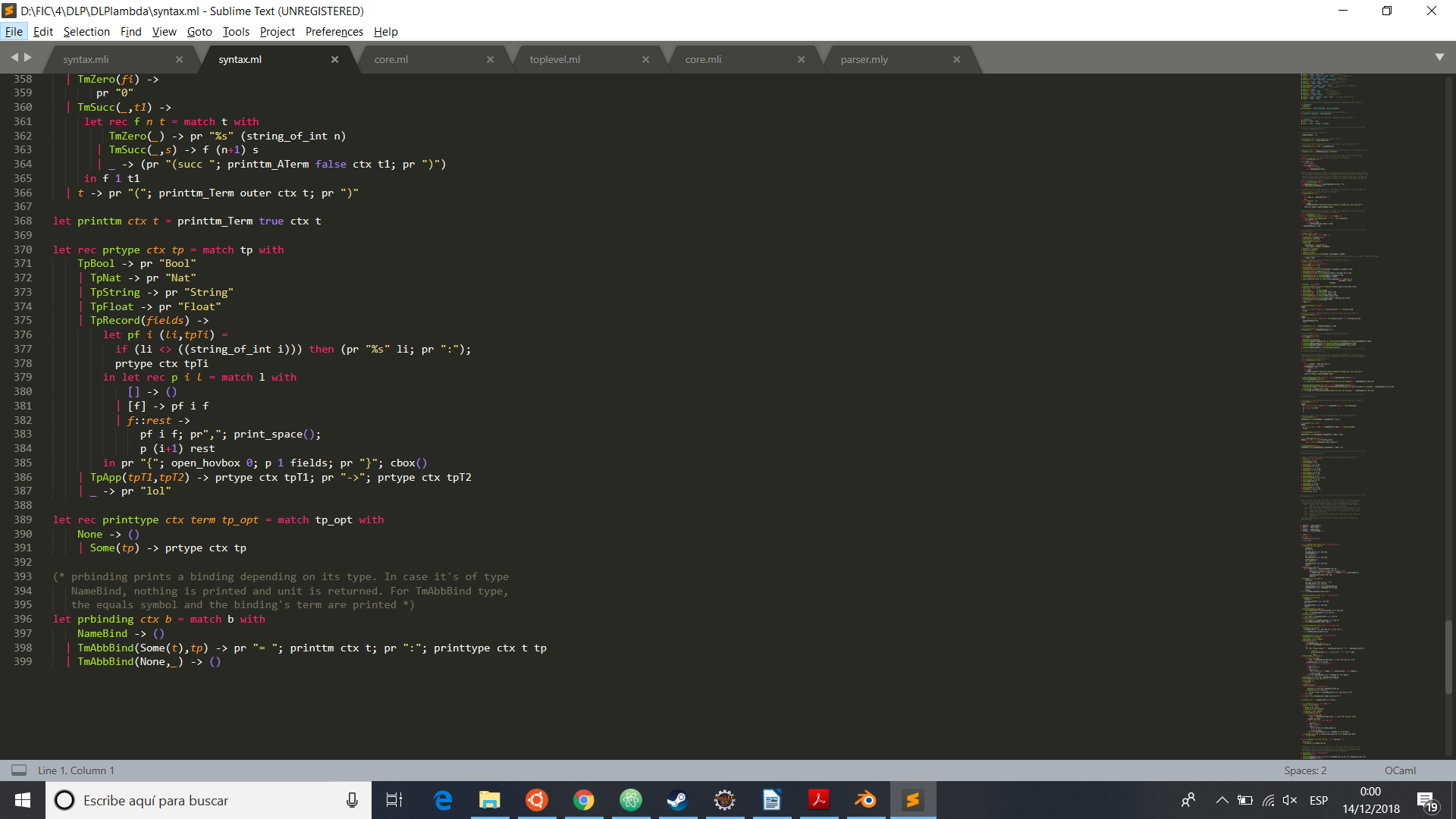
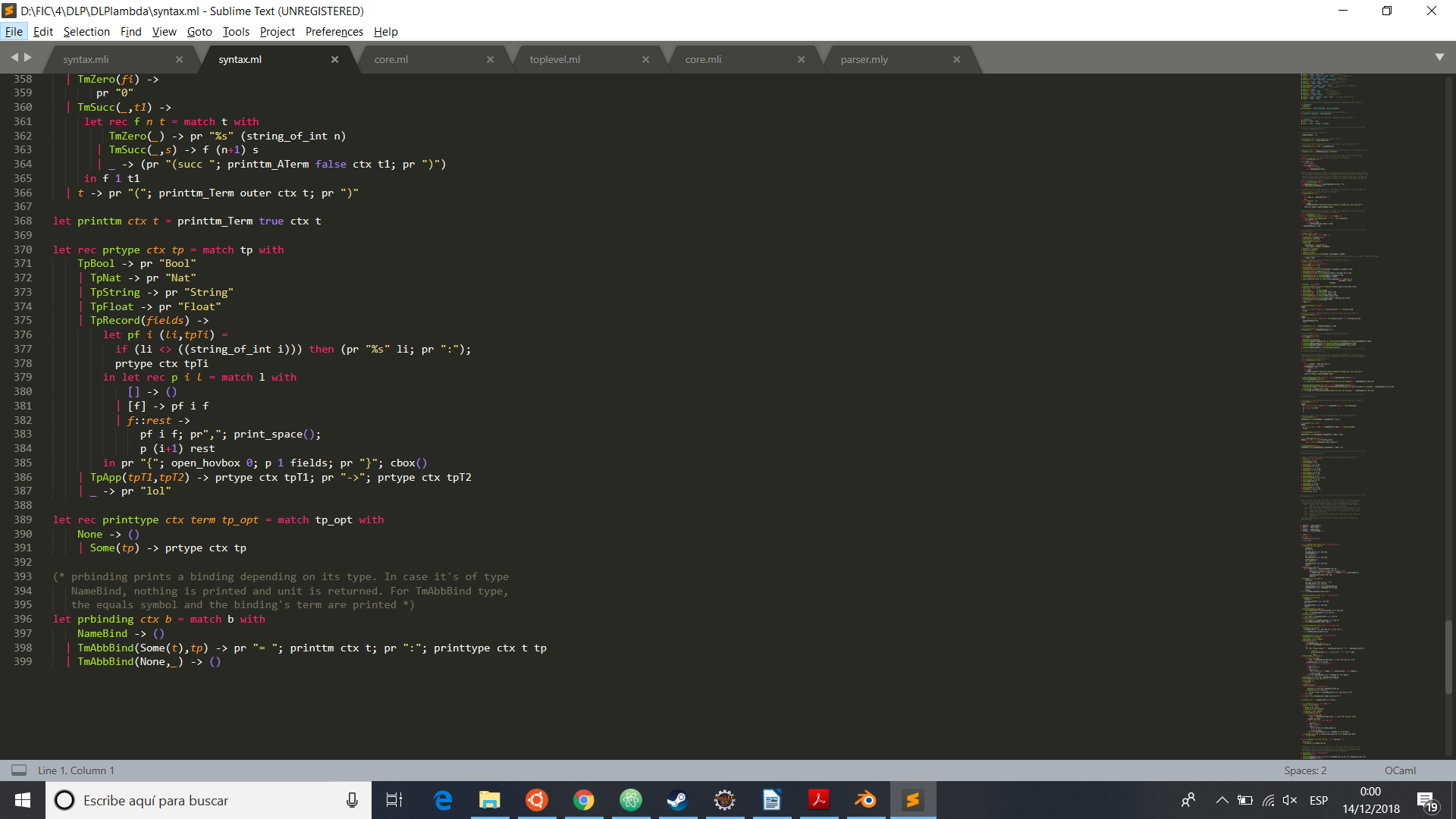


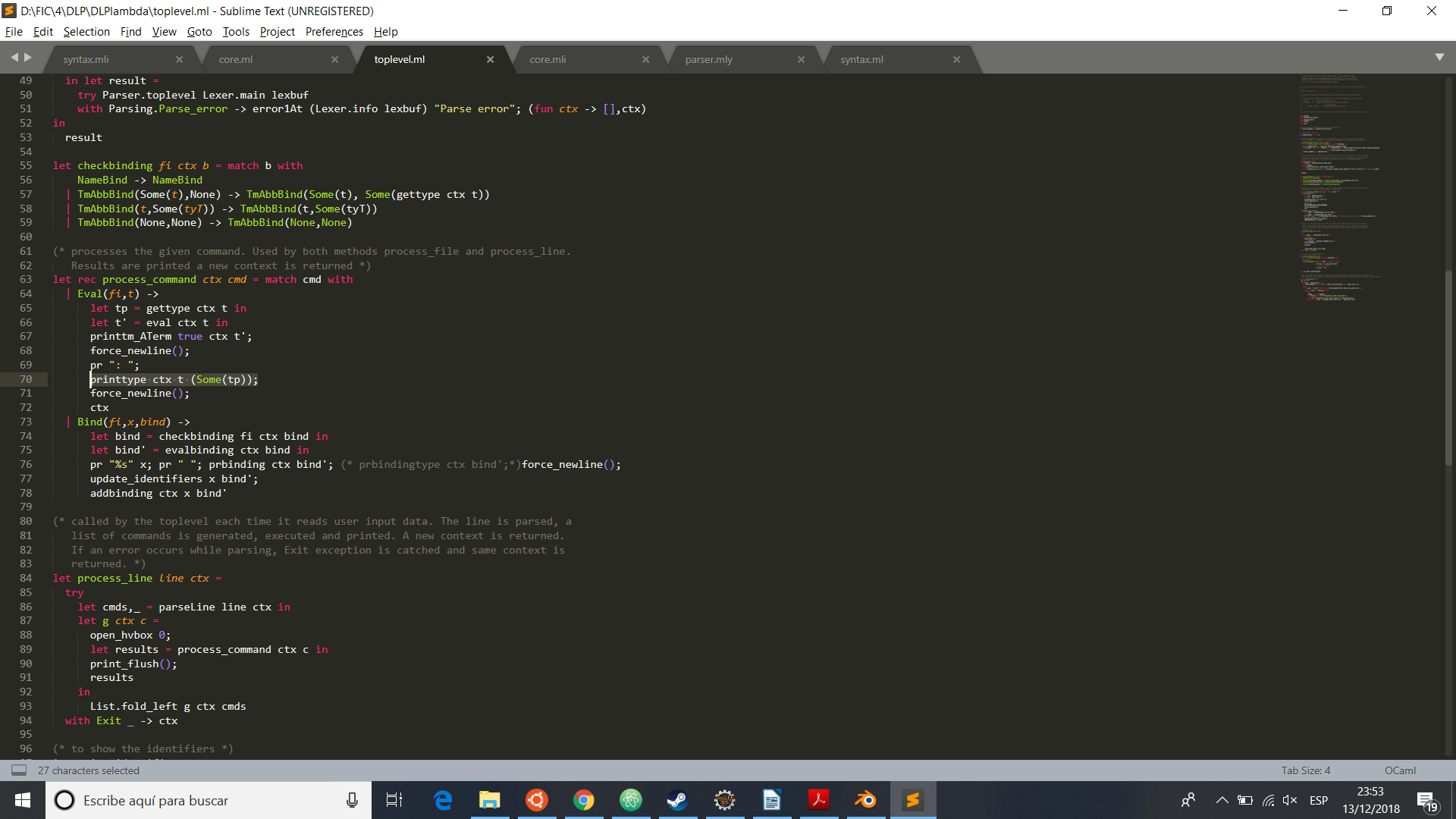
Después, añadimos al archivo core.ml (y a su correspondiente interfaz de módulo core.mli) la función ‘gettype’ (a la que nos referiremos durante la realización de todos los ejercicios), encargada de realizar las comprobaciones de tipo pertinentes. En este caso, además de comprobar los tipos simples derivados de los términos base (TmTrue, TmFalse y TmZero) de los tipos mencionados, también comprobaremos los tipos de las ‘built-in functions’ que están en el intérprete que se ha dado (TmSucc, TmPred, TmIsZero, TmIf).



Se omite, pero también hemos añadido la interfaz de esta función en el archivo core.mli, para su uso en el intérprete de alto nivel.

Acto seguido, definimos la función de printtype en el archivo syntax.ml para comprobar, finalmente, que la comprobación de tipos la hace correctamente y nos la muestra por terminal. A mayores, en el archivo ‘toplevel.ml’, retocamos la función process\_command añadiéndole la comprobación de tipos y la llamada al muestreo de tipos por pantalla.





Y de esta forma, ya tendríamos nuestro intérprete de tipos básicos para elementos mínimos de los tipos Bool y Nat.

## Ejemplos de Uso

Para comprobar el correcto funcionamiento de estas funcionalidades, además de anexar comandos de ejemplo que cubrirían todos los posibles casos de uso, mostraremos unos ejemplos y los explicaremos en este apartado, para cada ejercicio.

* Incorporación de TpBool y comprobación de TmIf:

>> true; (Análogamente, false;)

true: Bool

>> if true then true else false;

true: Bool

>> if 2 then true else false;

Error: La condición no es de tipo Bool

>> if true then 2 else false;

Error: Tipos diferentes en las ramas del If-Then-Else

* Incorporación de TpNat y comprobación de TmZero, TmIsZero, TmSucc, TmPred:

>> 2;

2: Nat

>> succ 2;

3: Nat

>> succ true; (Análogamente, funciona igual para pred).

Error: Expected value of type Nat for succ

>> succ (pred 10);

10: Nat

>> if true then iszero(succ 0) else iszero(pred 1);

false: Bool

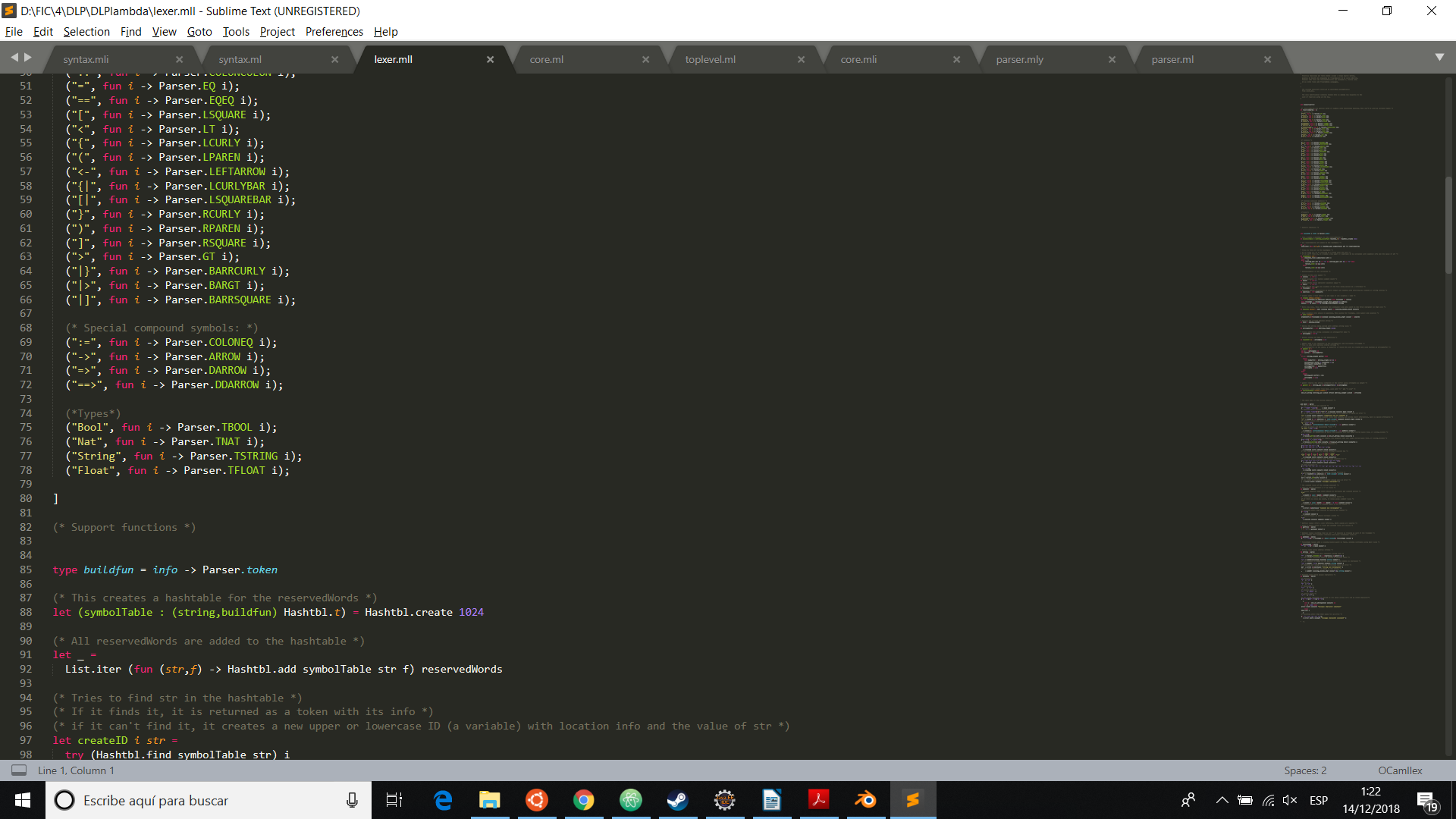
# Ejercicio 2

## Funcionalidades Solicitadas

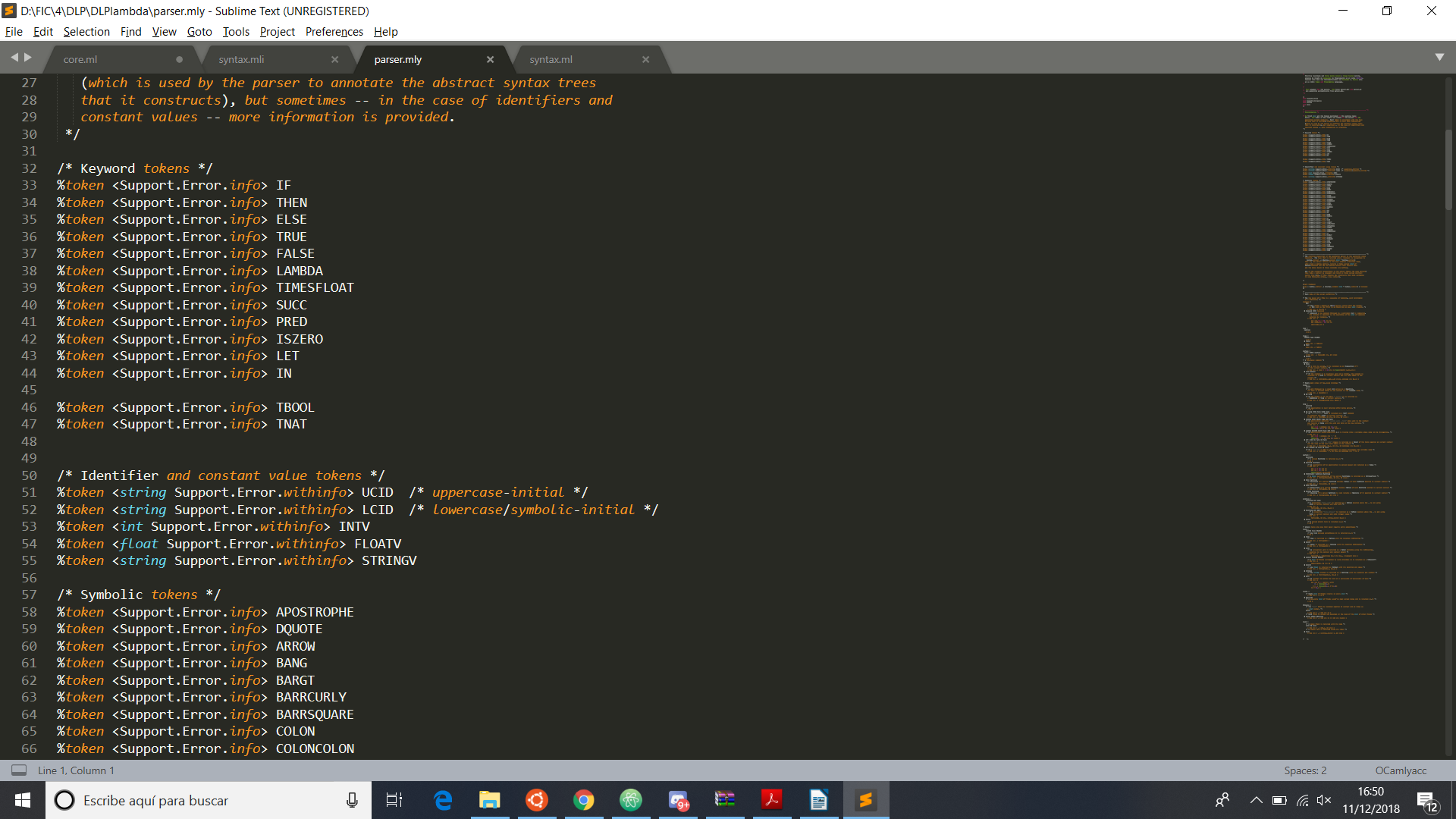
El comportamiento buscado en este ejercicio es que el nuevo intérprete no admita definiciones de funciones cuyos parámetros no vayan tipados de manera explícita.

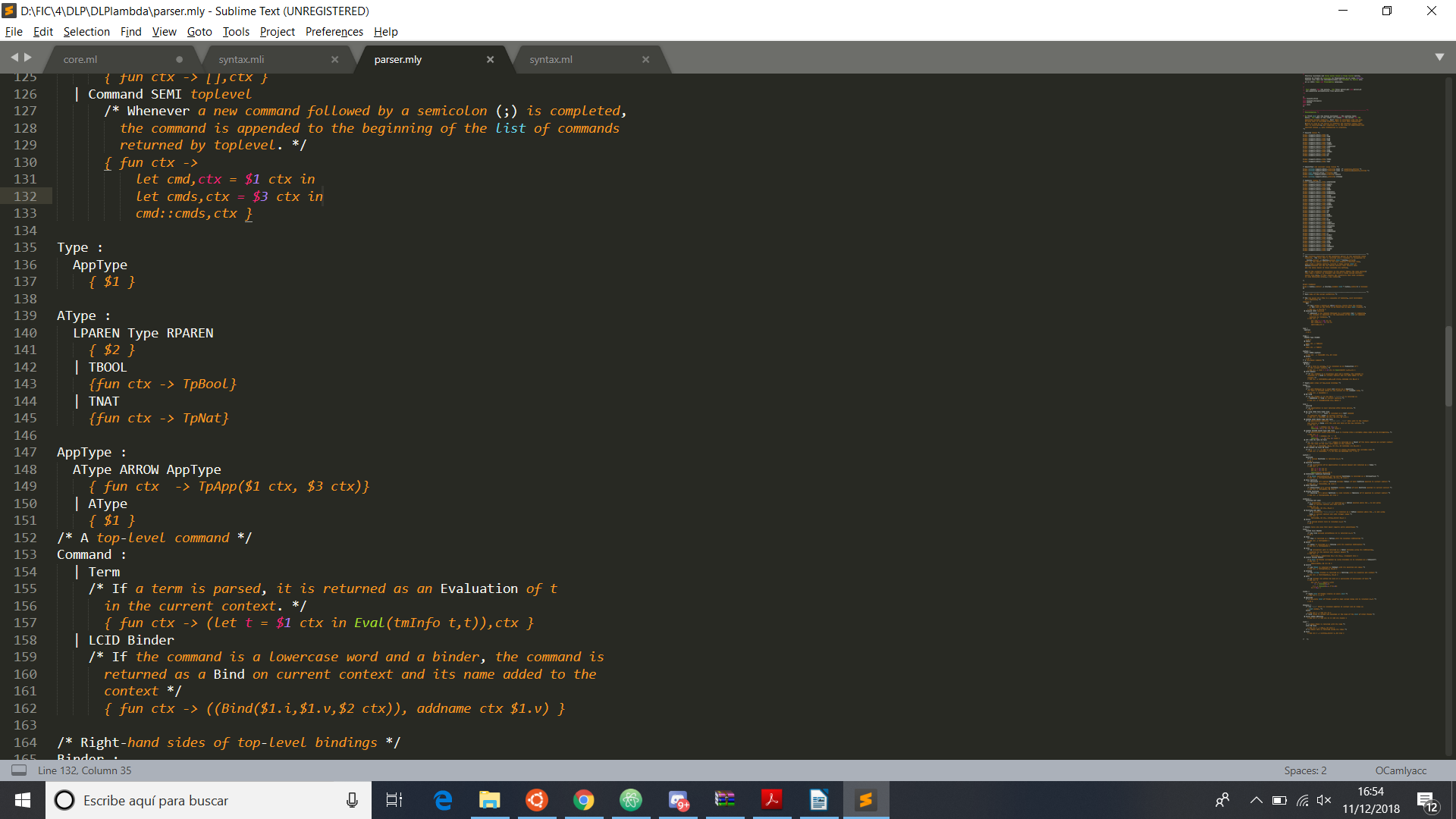
## Cambios Realizados

Para cambiar esto, lo primero que realizamos es un cambio en el lexer.mll, añadiendo los tipos que ya tenemos para poder pasarlos como entrada en la definición de funciones:

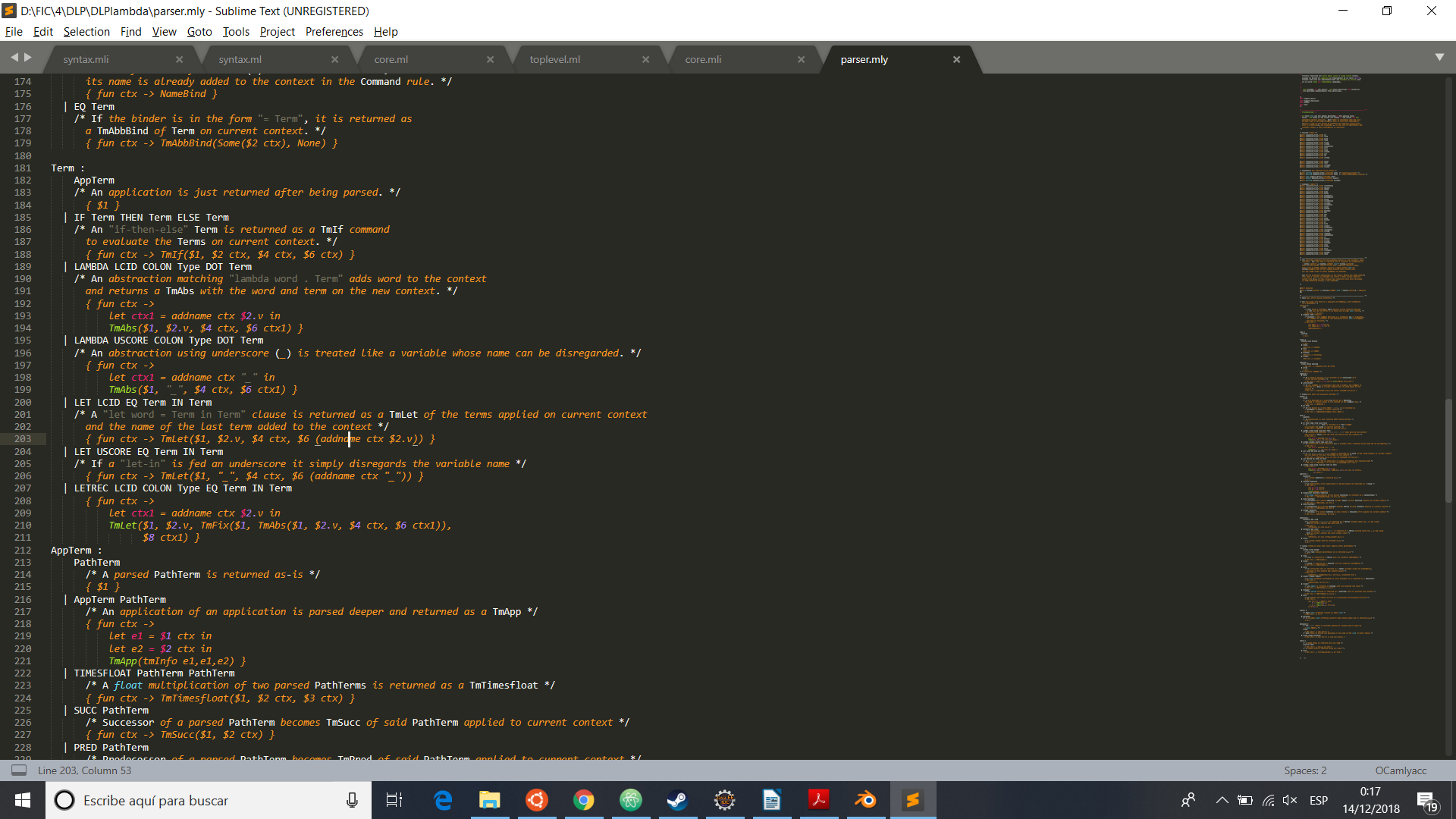


Asimismo, añadimos en el archivo parser.mly los elementos que reciban los tipos básicos que ya tenemos en el analizador léxico, y añadimos las reglas tanto de definición de tipos como aquellas que sirven para crear las lambda-abstracciones y los tipos de aplicación:



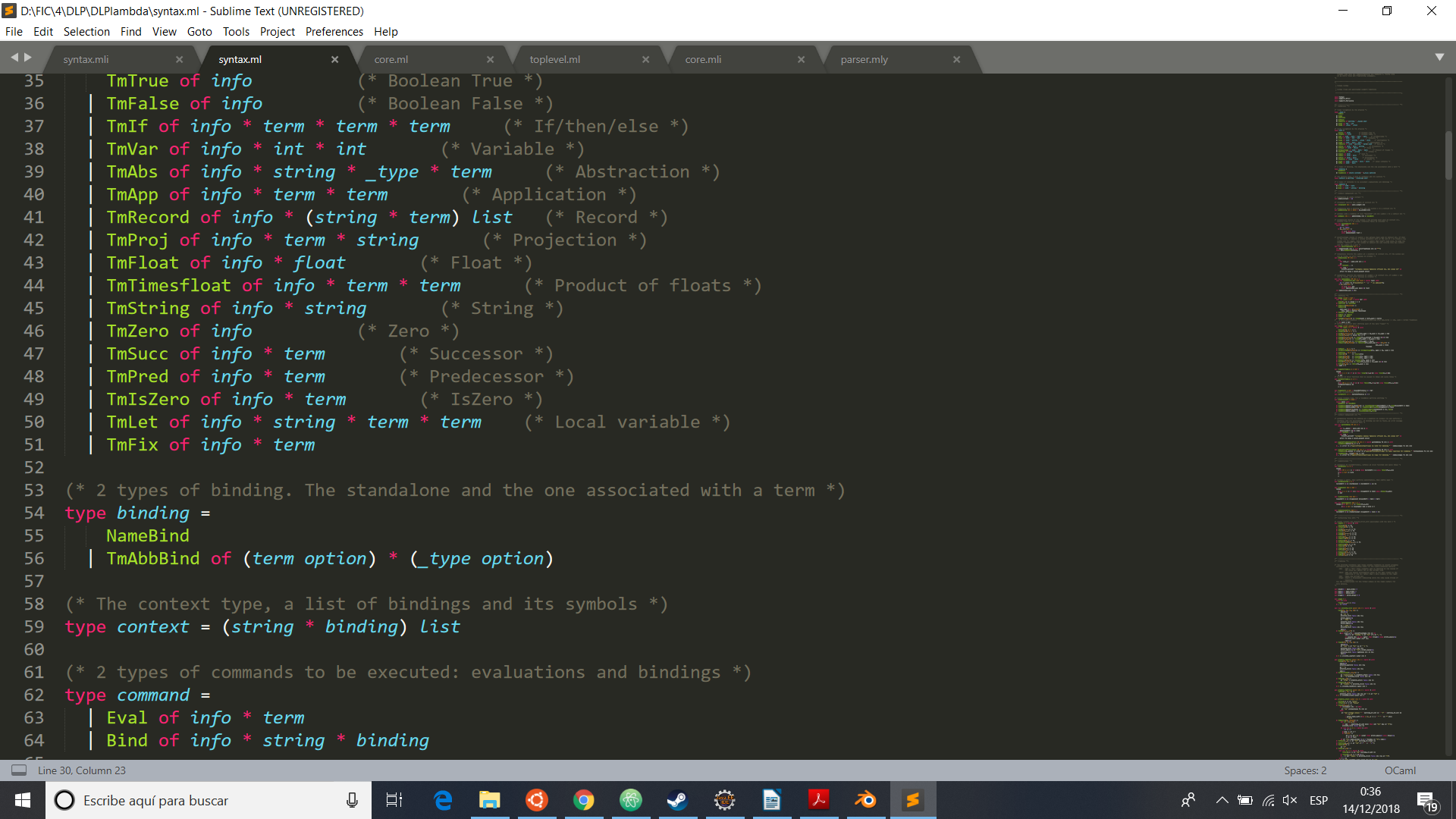


Una vez realizado esto, modificamos las reglas de las lambda-abstracciones:



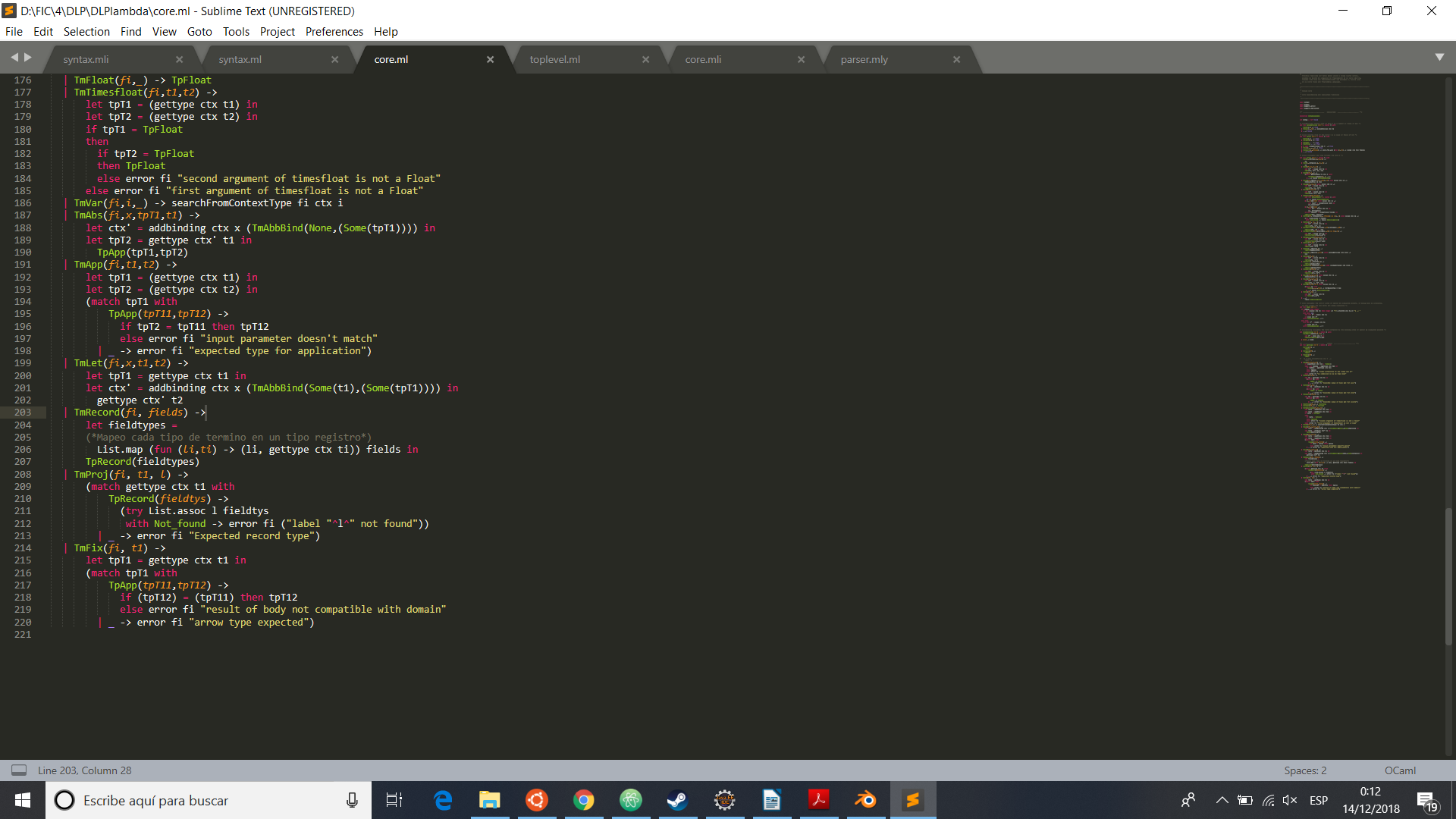
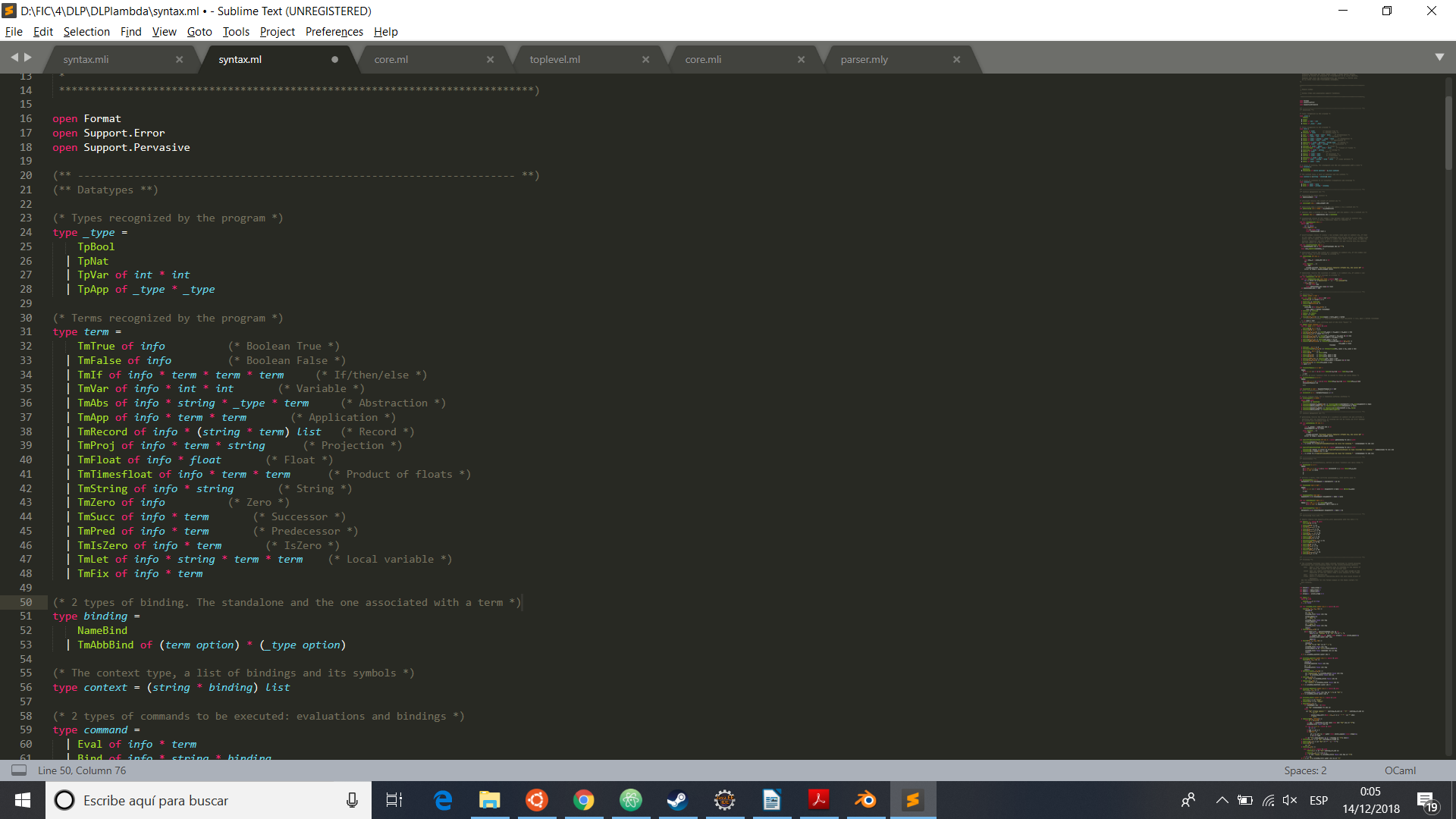
Añadiéndole el ‘:’ y el tipo, para definirlo de forma explícita en las funciones.

Después, a mayores, cambiaremos el tipo del binding que hay en ‘syntax.ml’ para añadirle el tipo a los términos (o a los parámetros de las lambda-abstracciones):



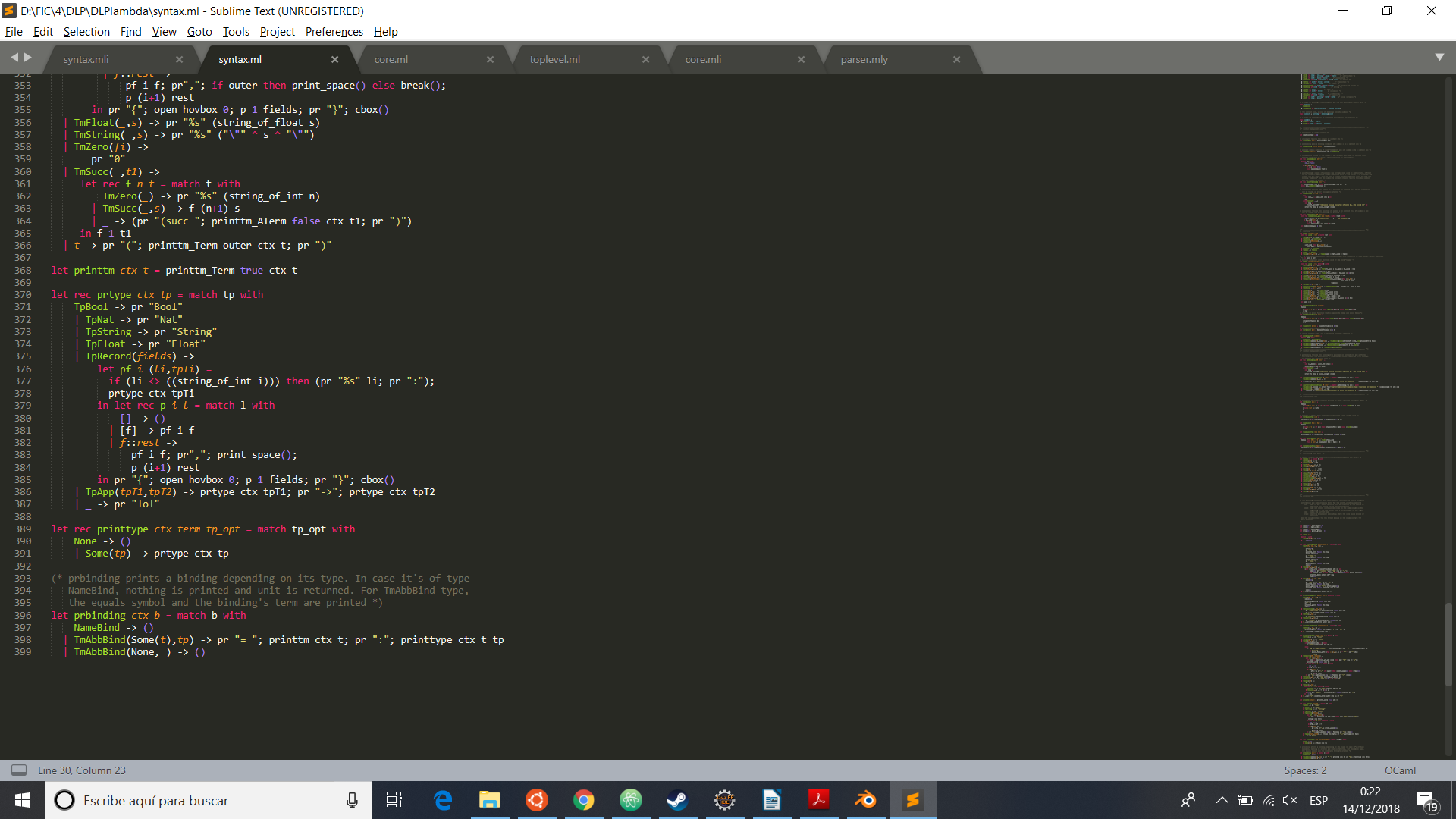
De esta forma, admitimos 4 posibles tipos de binding, según estén término y tipo expresados, alguno de ellos, o ninguno (este último caso no se da en el intérprete, pero se contempla en el resto del código para evitar errores). El binding asociado a la lambda-abstracción solo tendrá el tipo, y aprovecharemos esto para añadirle el tipo a las asignaciones y para la instrucción Let (TmLet).

Para continuar, añadimos las reglas de tipado en nuestra función gettype, y los términos correspondientes a las aplicaciones y a las variables en el archivo ‘syntax.ml’:



Para asegurarnos de que el contexto realiza bien estas operaciones, añadiremos una operación análoga al termShifting definida en el archivo ‘syntax.ml’ pero para los tipos que guardemos en el contexto, manteniendo la consistencia a la hora de definir bindings y evitando perder la referencia a ellos. Se omite debido a que su utilidad es importante pero no aporta nada diferente a lo que se proporciona en el código.

Para poder imprimir correctamente el contenido de los bindings por pantalla, modificamos una función que coge el tipo de los bindings y todos estos cambios los añadimos al ‘process\_command’ del toplevel.ml:



Y con esto ya estaría implementado el tipado explícito para la definición de funciones, así como el uso de asignaciones y del término TmLet para manejar el contexto con tipos.

## Ejemplos de Uso

Para la comprobación de las abstracciones, lo principal es saber si, con la implementación dada, el resto de cambios de nuestro código siguen siendo consistentes con lo que acabamos de introducir. Así pues, entremezclaremos distintos términos de distintas formas para comprobar que todo se realiza correctamente:

* Comprobación del funcionamiento del tipo explícito en las lambda-abstracciones:

>> lambda x:Bool. x;

(lambda x.x): Bool->Bool

>> lambda \_:Nat. true;

(lambda \_.true): Nat->True

>> lambda x:Bool. if x then 10 else 20;

(lambda x. if x then 10 else 20): Bool -> Nat

>> lambda x:Bool. (lambda y:Nat. if x then y else succ y);

(lambda x. (lambda y. if x then y else succ y): Bool->Nat->Nat

* Comprobación del funcionamiento de la aplicación:

>> (lambda x:Bool. if x then 10 else 20) true;

10: Nat

>> (lambda x:Bool. if x then 10 else 20) 10;

Error: Input parameter doesn’t match

>> 10 20;

Error: Expected arrow type

>> (lambda x:Bool. lambda y:Nat. if x then y else succ y) false 20;

21: Nat

* Comprobación del correcto funcionamiento de la asignación, y de la asignación local mediante la instrucción Let, así como del manejo del contexto;

>> x = 10;

x = 10: Nat

>> x;

10: Nat

>> x = lambda x:Bool. if x then 10 else 12;

x = lambda x.if x then 10 else 12: Bool->Nat;

>> x true;

10: Nat

>> let x = 10 in (succ x);

11: Nat

>> let y=12 in let a=10 in lambda x:Nat. lambda a:Nat. if true then (succ a) else (pred y);

lambda x. lambda a. if true then (succ a) else (pred y): Nat->Nat->Nat;

>> fun = let y=12 in let a=10 in lambda x:Nat. lambda a:Nat. if true then (succ a) else (pred y);

fun = lambda x. lambda a. if true then (succ a) else (pred y): Nat->Nat->Nat;

>> fun 10 12;

13: Nat

>> lambda x:Bool. lambda x:Nat. pred x;

lambda x:Bool. lambda x’.pred x’: Bool->Nat->Nat

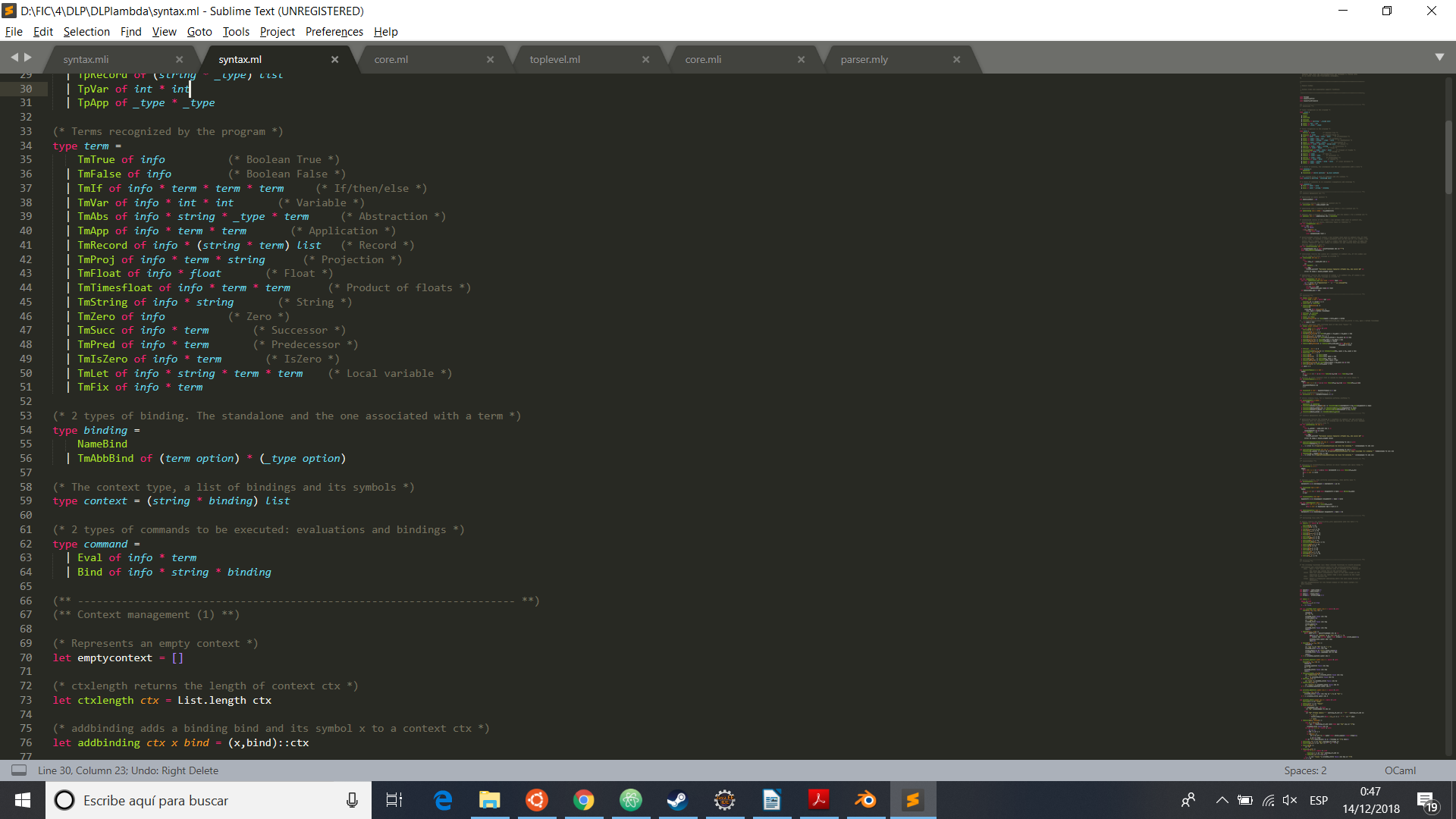
# Ejercicio 3

## Funcionalidades Solicitadas

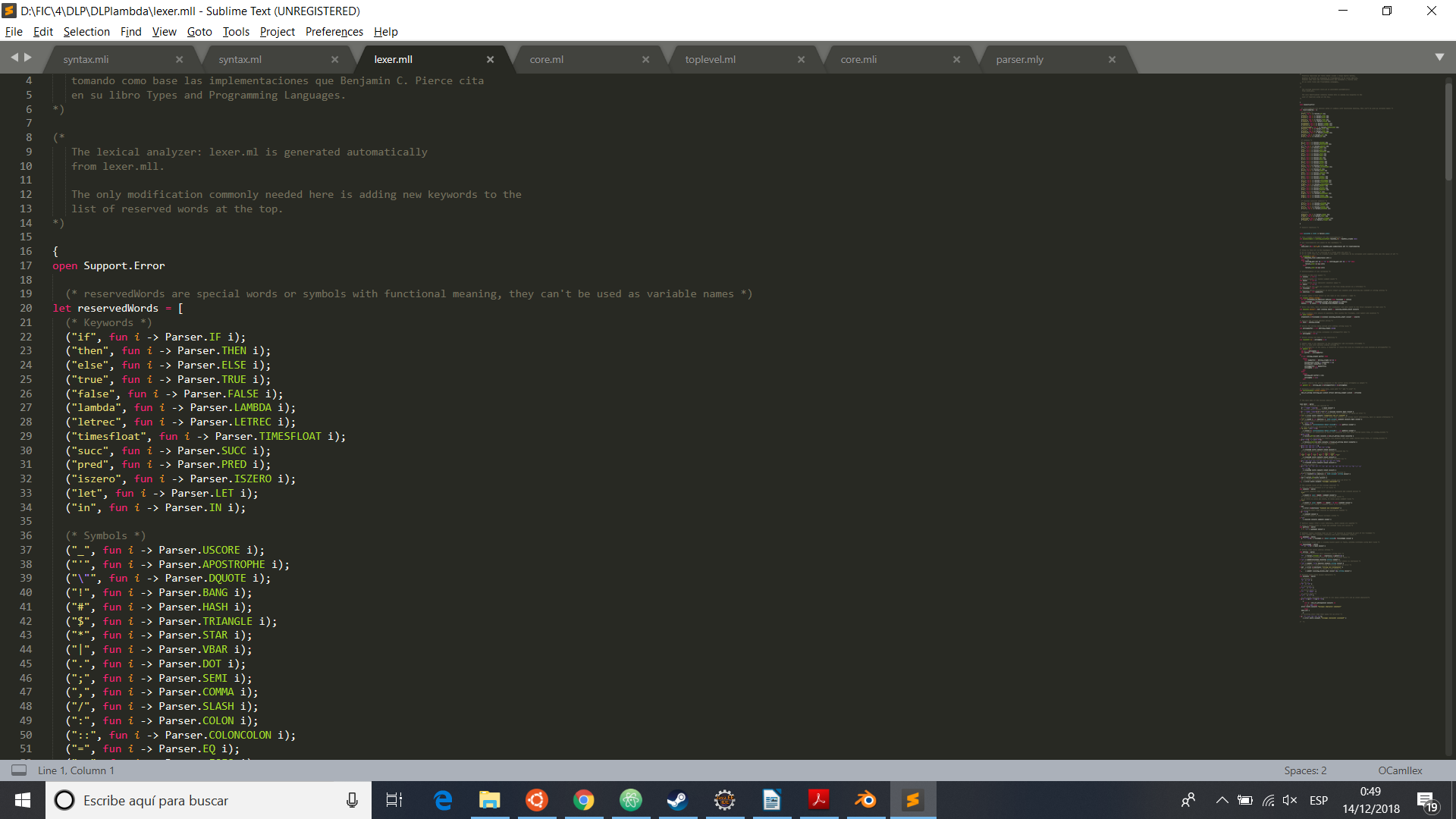
El tercer apartado de la práctica es incorporar al intérprete un combinador de punto fijo interno (fix), de tal forma que se puedan declarar este tipo de funciones mediante definiciones recursivas directas.

## Cambios Realizados

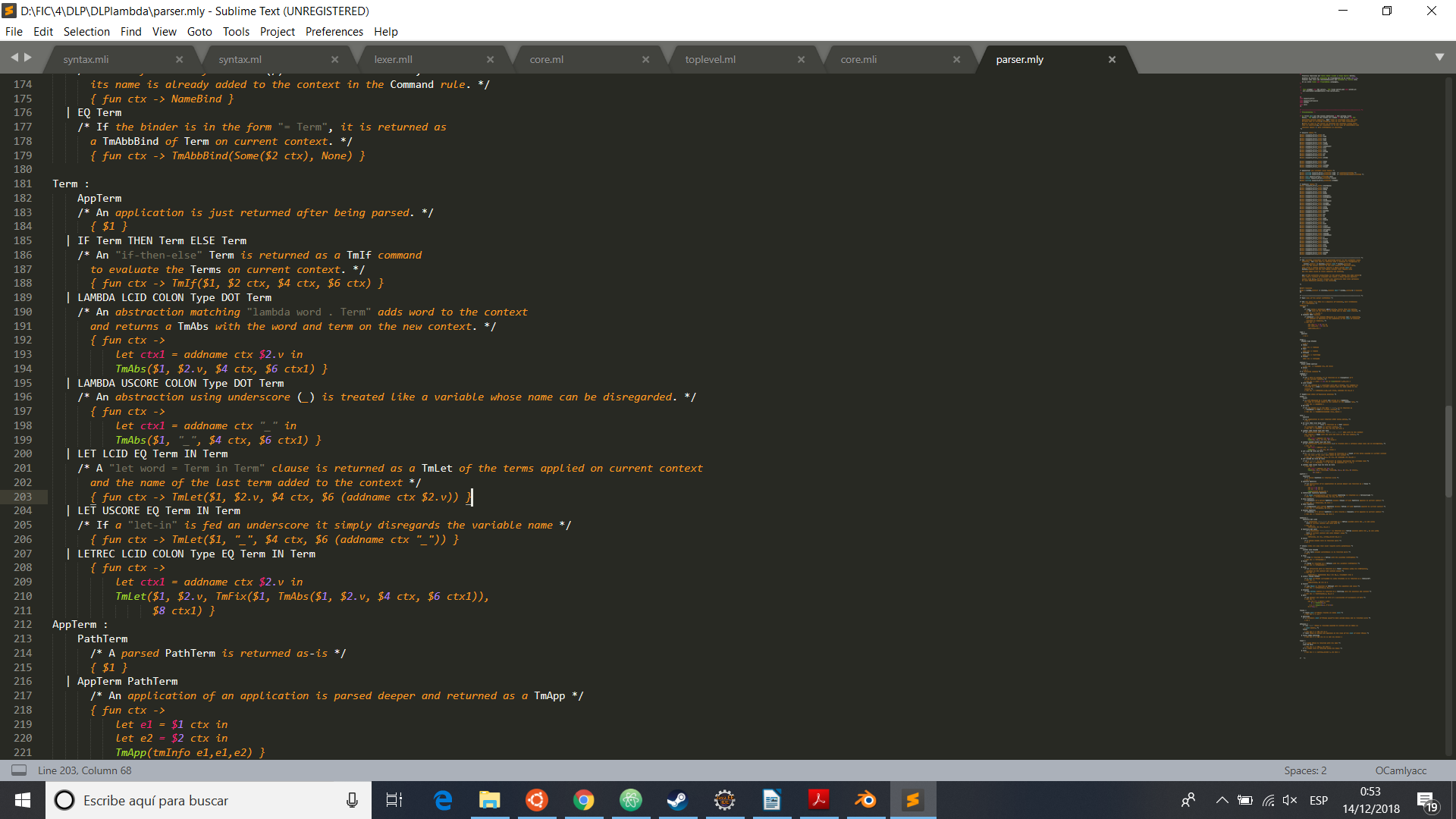
Para afrontar este ejercicio, comenzamos definiendo TmFix en el archivo ‘syntax.ml’ para añadir el operador de punto fijo y poderlo utilizar para habilitar la recursión:



En el archivo ‘lexer.mll’ añadimos la palabra reservada “letrec” para poder definir funciones recursivas:



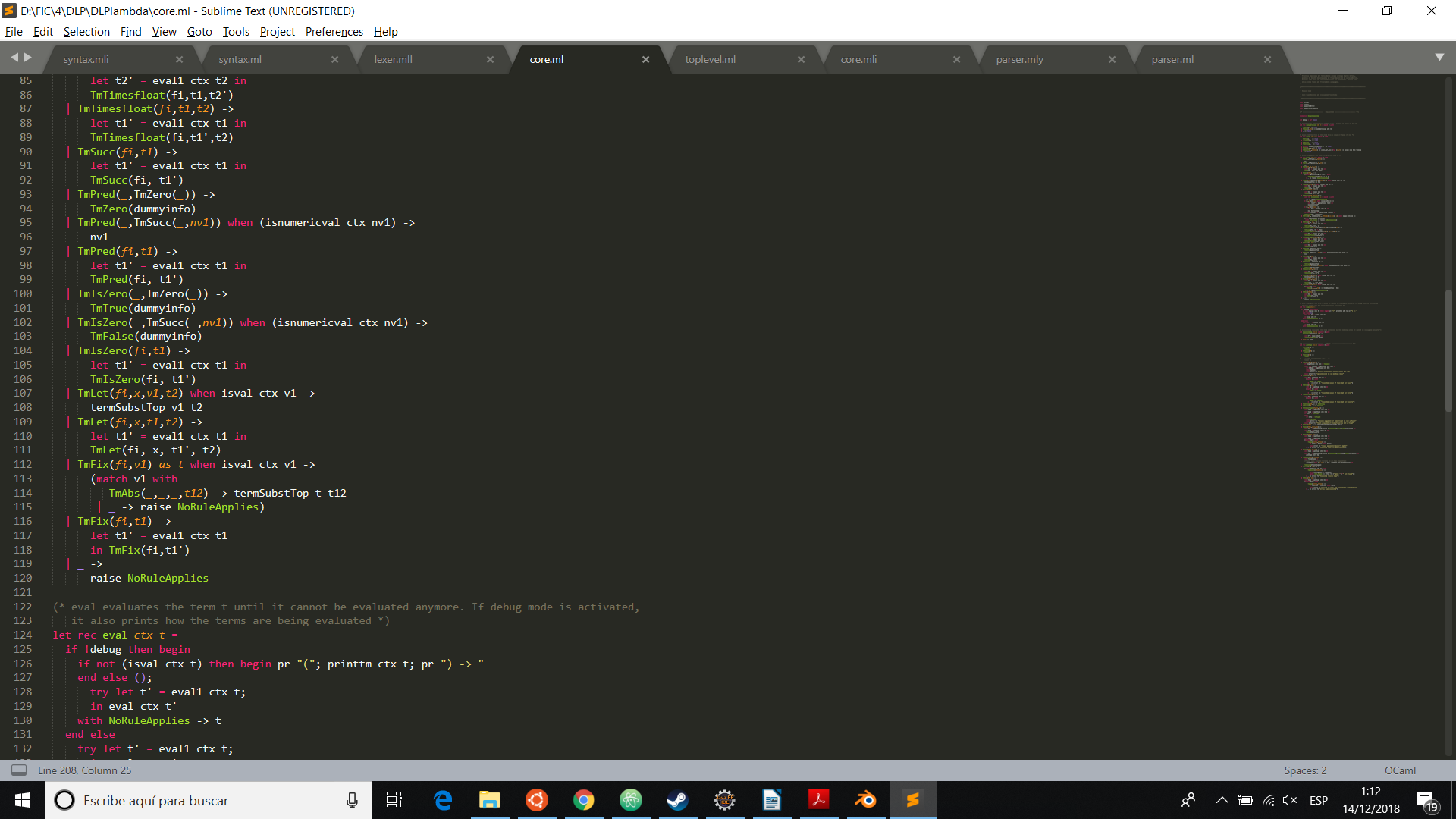
A continuación, añadimos al parser.mly la regla que nos permitirá la recursión dentro de una abstracción:



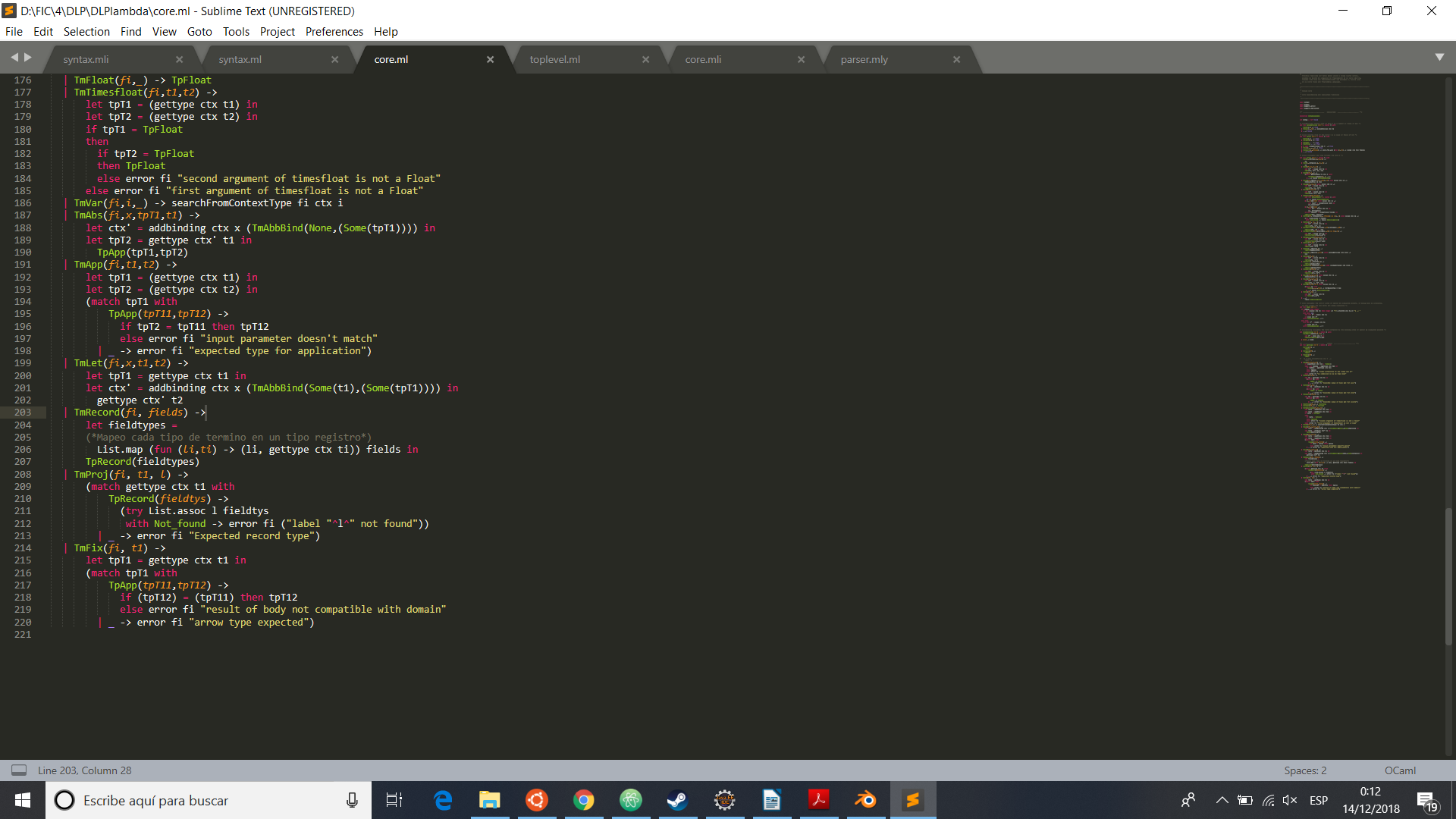
El cual es una combinación de la sentencia Let, junto al uso del término del operador fix que lleva dentro la abstracción a la que le tiene que aplicar la recursión.

Ahora, tendremos que tocar el archivo ‘core.ml’ para añadir la regla de evaluación de fix y poder también deducir el tipo en la función de obtención de tipo:

Eval:



Gettype:



## Ejemplos de Uso

Para comprobar la correcta implementación del combinador de punto fijo, realizaremos una función de recursión de suma mediante letrec, y trataremos de concatenar otra función recursiva dentro de la primera:

* Comprobación del correcto funcionamiento de la instrucción letrec y de TmFix:

>> letrec sum:Nat -> Nat -> Nat = lambda n:Nat.lambda m:Nat. if iszero n then m else succ(sum (pred n) m) in sum 2 3;

5: Nat

>> letrec sum\_1:Nat -> Nat -> Nat = lambda n:Nat.lambda m:Nat. if iszero n then m else succ(sum\_1 (pred n) m) in letrec sum\_2:Nat->Nat->Nat = lambda n:Nat.lambda m:Nat. if iszero n then m else succ (sum\_2(pred n) m) in sum\_2 (sum\_1 4 5) 9;

18: Nat

# Ejercicio 4

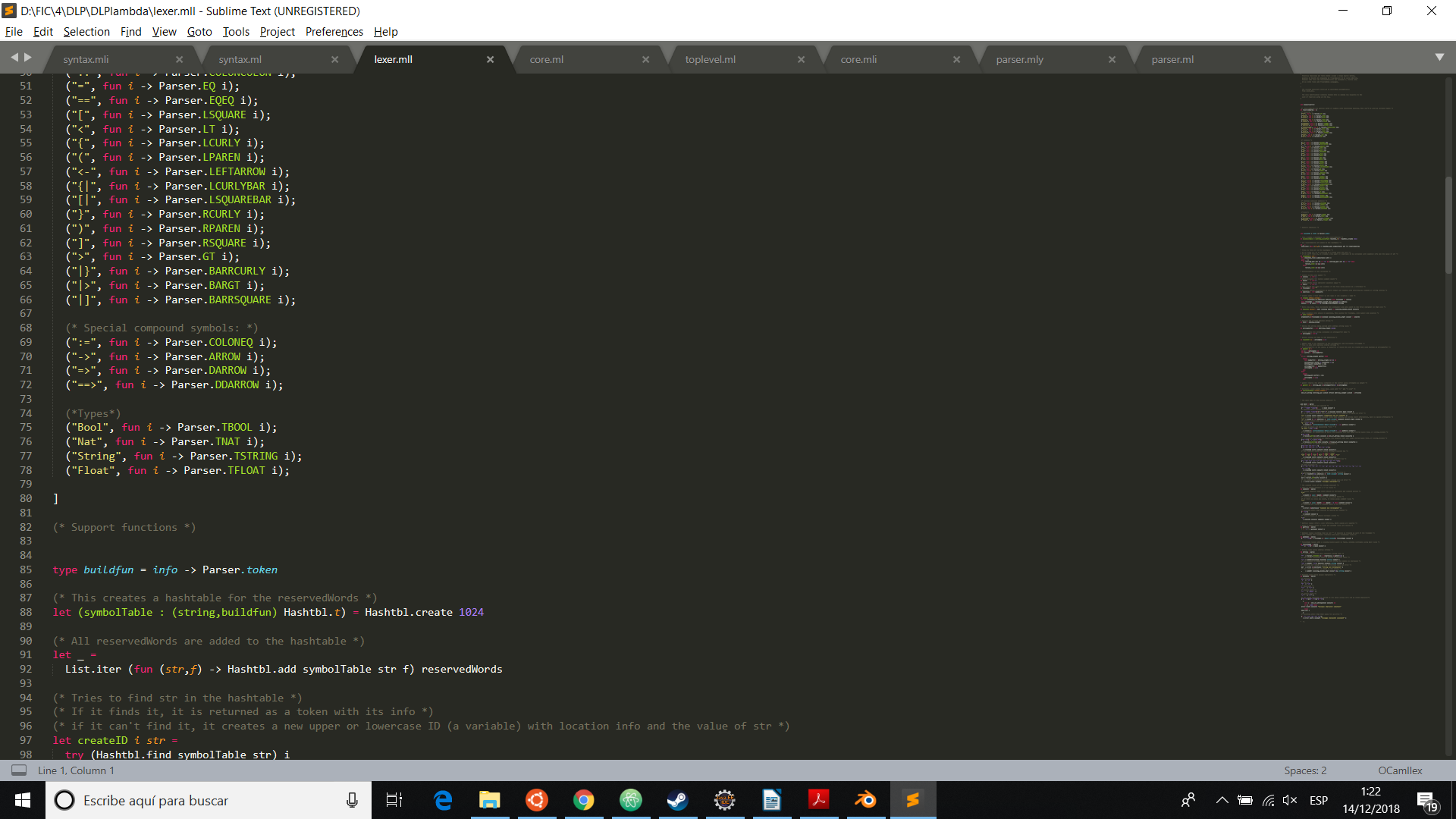
## Funcionalidades Solicitadas

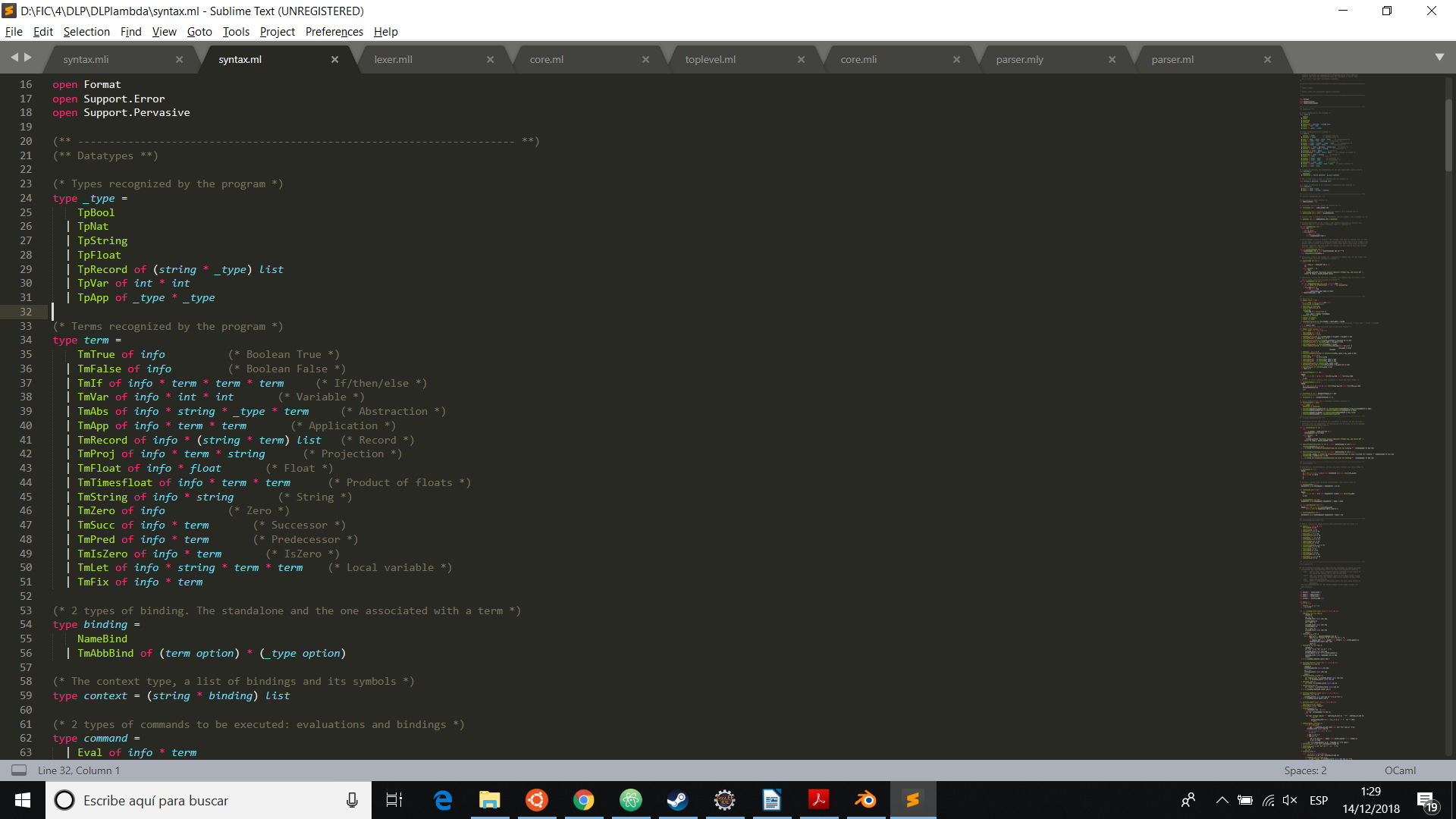
El último apartado de la práctica consiste en gestionar otros tipos adicionales como pueden ser los tipos String, Float o Record.

Para ello las opciones planteadas son, rechazar estas expresiones, conservar el comportamiento original o bien definir nuevos tipos de datos; En nuestro caso decidimos definir nuevos tipos de datos.

## Cambios Realizados

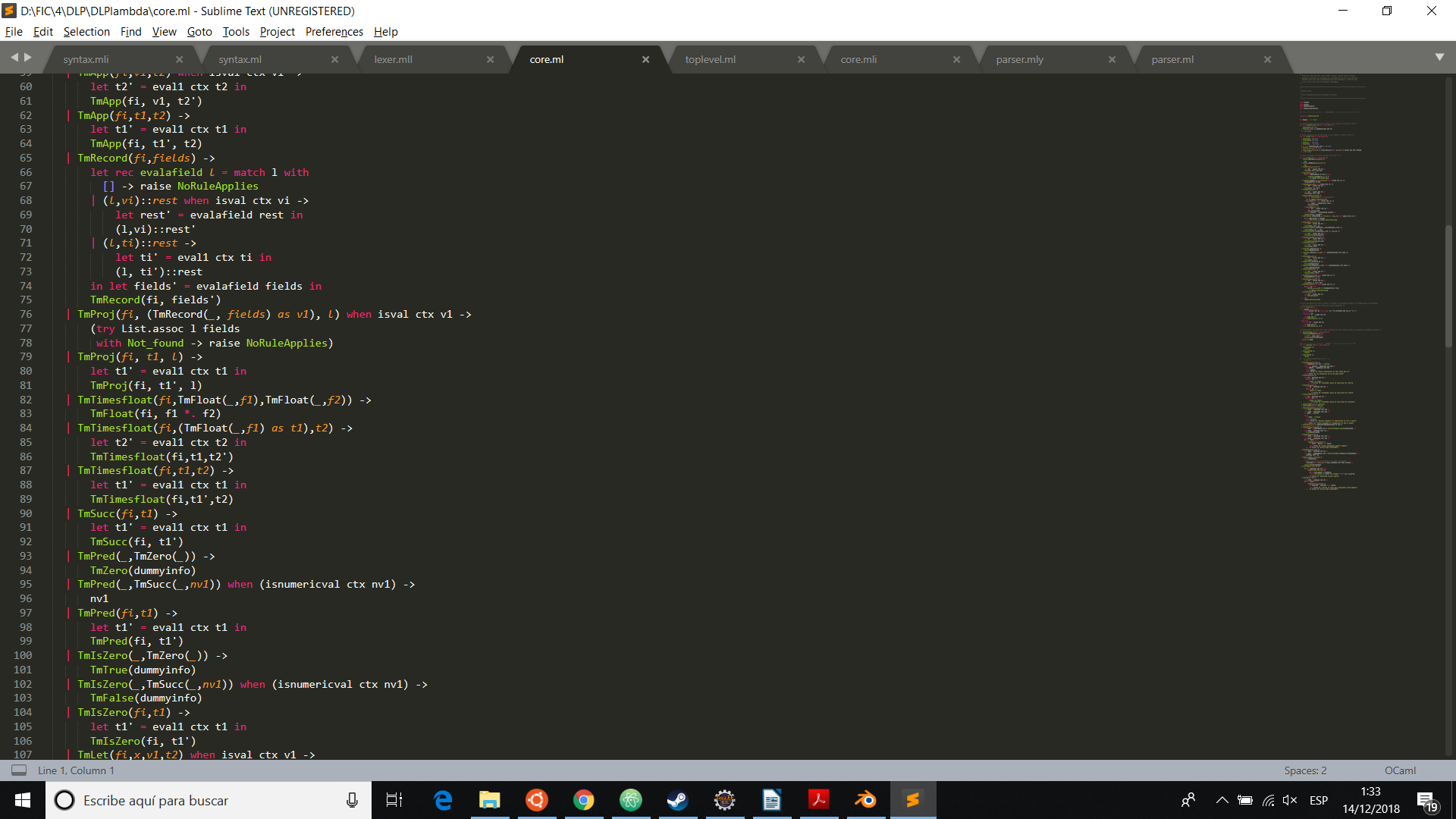
En este apartado comenzamos con los tipos String y Float que son los más similares a los tipos Nat y Bool definidos previamente, para ello definimos los nuevos tipos en el lexer.mll, aceptándolos en el parser.mly y creando los tipos en el syntax.ml:





Del tipo record hablaremos más adelante, por ser un poco más complejo.

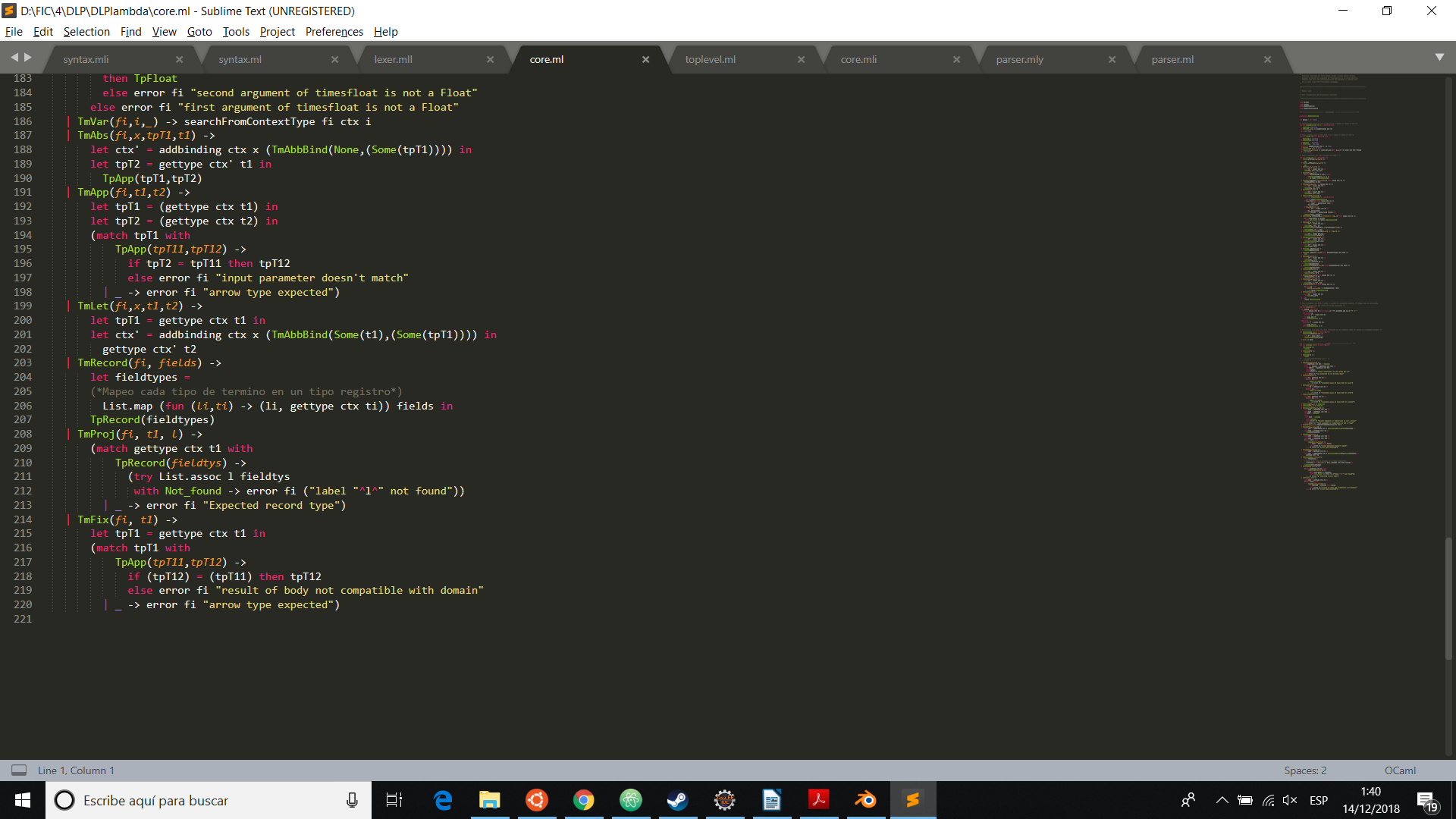
Continuamos añadiendo al core.ml las reglas de obtención de tipos, siendo los tipos básicos nuevos de la misma tónica que los tipos Bool y Nat. A mayores, tenemos en cuenta la primitiva ‘TmTimesFloat’, que consiste en una multiplicación de dos TmFloat.



Por último, modificamos la función de impresión de tipos en el archivo ‘syntax.ml’:



Ahora, les toca el turno a los registros (y el correspondiente término de la proyección sobre éste). Para eso definimos el tipo de registro (TpRecord, ya mostrado en la imagen anterior del archivo syntax.ml). Este tipo de registro guarda una lista de tipos, donde cada tipo se refiere a cada uno de los elementos del registro, guardando la etiqueta de cada uno de los campos y su tipo. Ahora, falta añadir la regla de obtención de tipo en nuestra función gettype:



Para el record, utilizamos la función de evaluación elemento a elemento y la mapeamos sobre una lista que contendremos en el TpRecord. Para la proyección, buscaremos la etiqueta en el nombre del registro, y si existe, devuelve el tipo asociado, calculado previamente.

Con esto ya tendríamos completo nuestro intérprete de lambda cálculo con recursividad implementada y todos los tipos que se nos han dado para su uso.

## Ejemplos de Uso

* Comprobación del correcto funcionamiento del tipo Float y del tipo String:

>>10.2;

10.2: Float;

>>timesfloat 10.0 2.5;

25.0: Float;

>>mult = lambda x:Float. lambda y:Float. timesfloat x y;

mult = (lambda x. lambda y. timesfloat x y): Float-> Float-> Float

>> timesfloat true 10.0;

Error: first argument of timesfloat is not a Float

>> timesfloat 10.0 true;

Error: second argument of timesfloat is not a Float

>> “Hola Mundo”;

“Hola Mundo”: String

>> let x=”hola” in x;

“hola”: String

>>echo = lambda x:String. x;

echo = (lambda x.x): String -> String

>> echo “10”;

“10”: String;

* Comprobación del correcto funcionamiento del tipo Record y del tipo Proj:

>> {};

{}: {}

>> rec = {1,”hola”,10.0,true};

rec = {1,”hola”,10.0,true}: {Nat, String, Float, Bool}

>> rec.1;

1:Nat

>> rec.etiqueta

Error: label etiqueta not found

>> rec = {nat=1,string=”hola”,10.0,bool=true};

rec={nat=1, string=”hola”, 10.0, bool=true}: {nat:Nat ,string:String ,Float, bool:Bool}

>> rec.nat;

1: Nat;

>> rec.1;

Error: label 1 not found;

>> rec.3;

10.: Float

>> rec = {lambda x:Nat. iszero x, let x=10 in succ x};

rec = {lambda x. iszero x, 11}: {Nat->Bool, Nat}

>> rec.1;

(lambda x. iszero x): Nat->Bool

>> rec.1 0;

true: Bool

>> rec = {lambda x:Nat. iszero x, let x=10 in succ x, letrec sum:Nat->Nat->Nat = lambda m:Nat. lambda n:Nat. if iszero n then m else succ (sum (pred n) m) in sum 10 20};

rec = {lambda x. iszero x, 11, 30} : {Nat->Bool, Nat, Nat}