Nota. Este documento es un fragmento del Seminario Final de Licenciatura en Informática de Federico Lochbaum en la Universidad Nacional de Quilmes.

1. Sintaxis léxica

En esta sección se describe la sintaxis léxica del lenguaje Ñuflo.

1.1. Lexemas

El programa fuente es una cadena de caracteres Unicode que se asume representada en la codificación UTF-8. Un programa Ñuflo se segmenta como una secuencia de "lexemas" o *tokens*. Cada lexema viene acompañado de su nombre en `MAYÚSCULAS`, al que se hará referencia en la gramática formal.

Blancos y comentarios.

En el proceso de análisis léxico, los caracteres en blanco sirven como delimitadores de lexemas, pero no componen lexemas por sí mismos. Los caracteres en blanco son: espacios (`' '`), tabs (`'\t'`), saltos de línea (`'\n'`) y retornos de carro (`'\r'`).

Se ignoran también los comentarios en la entrada. Existen dos tipos de comentario: comentarios de línea comenzados en `--` que se extienden hasta el siguiente salto de línea (`'\n'`), y comentarios multilínea comenzados en `{-` que se extienden hasta el siguiente `-}`. Los comentarios multilínea pueden anidarse.

Además, el lenguaje Ñuflo es sensible a la identación. Los caracteres en blanco tienen relevancia para la aplicación de las reglas de *layout* que se describen en el siguiente apartado.

Literales.

Se distinguen tres tipos de literales:

- `INT`: identifica los literales numéricos. Un literal numérico se escribe como una secuencia no vacía de dígitos decimales, reconocidos por la expresión regular `[0-9]+`.
- `CHAR`: identifica los literales de caracter. Un literal de caracter se escribe como una comilla simple (`'`) seguida de un caracter, seguido de otra comilla simple. Por ejemplo, `'A'` identifica la letra A mayúscula. Los caracteres literales pueden incluir secuencias de escape. Se reconocen las siguientes secuencias de escape:

  `'\\'`: contrabarra (\).

  `'\''`: comilla simple (').

  `'\"'`: comilla doble (").

`'\a'`: *alert / beep / bell*.

`'\b'`: retroceso.

`'\f'`: salto de página.

`'\n'`: salto de línea.

`'\r'`: retorno de carro.

`'\t'`: tab.

`'\v'`: tab vertical.

- `STRING`: identifica los literales de cadena. Un literal de cadena se escribe como una comilla doble (`"`) seguida de una secuencia de caracteres, seguido de otra comilla doble. Las cadenas literales pueden incluir secuencias de escape como las descriptas arriba. Por ejemplo, `"hola\n"` representa la cadena "hola" seguida de un salto de línea.

Puntuación.

Las siguientes secuencias de caracteres se consideran signos de puntuación:

```
DOT         .
LPAREN      (
RPAREN      )
LBRACE      {
RBRACE      }
SEMICOLON   ;
```

Identificadores y palabras clave.

Un identificador se compone de una secuencia de caracteres consecutivos, cada uno de los cuales debe cumplir las siguientes condiciones:

- Debe ser un caracter Unicode imprimible.
- No debe ser un blanco (`' '`, `'\t'`, `'\n'`, `'\r'`).
- No debe ser un signo de puntuación (`'.'`, `'('`, `')'`, `'{'`, `'}'`, `';'`).

Notar que esto incluye caracteres alfanuméricos y símbolos (como `+`, `-`, `*`, etc.). Notar también que esto incluye la coma (`,`), de modo que "`a,b`" es un solo identificador y no es equivalente a la secuencia de dos identificadores "`a, b`" ni a la secuencia de tres identificadores "`a , b`". Además se imponen las siguientes dos condiciones:

- El identificador no puede contener dos guiones bajos (`_`) consecutivos. Así, por ejemplo, `if_then_else_` y `_*_` son identificadores válidos, mientras que `[__]` no lo es.

- El identificador no debe constar únicamente de dígitos decimales (pues en tal caso, se trataría de un `INT`).

Como excepciones se incorporan dos sinónimos, que permiten escribir algunos símbolos muy frecuentes de manera alternativa en ASCII o con símbolos Unicode especiales:

- "λ" (caracter `U+03bb`) es sinónimo de "\",
- "→" (caracter `U+2192`) es sinónimo de "`->`".

Además, los siguientes identificadores distinguidos son **palabras clave** del lenguaje Ñuflo:

`AS`	`as`
`CASE`	`case`
`CLASS`	`class`
`DATA`	`data`
`EQ`	`=`
`FRESH`	`fresh`
`IMPORT`	`import`
`IN`	`in`
`INFIX`	`infix`
`INFIXL`	`infixl`
`INFIXR`	`infixr`
`INSTANCE`	`instance`
`LAMBDA`	`\` *ó* λ
`LET`	`let`
`MODULE`	`module`
`OF`	`of`
`TYPE`	`type`
`WHERE`	`where`

Los identificadores que *no* son palabras clave corresponden al lexema `ID`.

1.2. Reglas de *layout*

El lenguaje Ñuflo es sensible a la identación. Algunas palabras clave, las llamadas palabras clave *offside*, determinan el inicio de un "bloque". Las palabras clave *offside* son las siguientes:

`let` `fresh` `of` `where`

El lexema que se encuentra en la entrada después de una palabra clave *offside* determina el nivel de identación del bloque. El bloque se inicia con un token

LBRACE ({) que puede estar escrito explícitamente en la entrada o implícito por la presencia de la palabra clave *offside*.

El bloque se mantiene abierto en tanto que todos los lexemas comiencen en una columna posterior a la indicada por el nivel de identación. El bloque se cierra en cuanto se encuentra un lexema que comienza en una columna estrictamente más chica que la indicada por el nivel de identación. El bloque se cierra con un token RBRACE (}) que debe estar explícito si y sólo si el token de apertura del bloque estaba explícito.

Además los bloques constan de elementos separados por el token SEMICOLON (;), que puede estar explícito, o implícito cada vez que un lexema comienza exactamente en la columna correspondiente al nivel de identación del bloque.

Por ejemplo, en el siguiente programa de entrada:

```
module A where
null list = case list of
              []      → True
              (_ : _) → False
main = print (null []) end
```

Se insertan los delimitadores LBRACE ({), RBRACE (}) y SEMICOLON (;) como sigue:

```
module A where {
null list = case list of {
              []      → True
            ; (_ : _) → False
            };
main = print (null []) end
}
```

1.3. Analizador léxico

La responsabilidad del analizador léxico o *lexer* es segmentar el código fuente del programa en lexemas, verificando que el programa respete las reglas impuestas por la sintaxis léxica y construyendo una representación intermedia del programa, entendido como una secuencia de tokens. Concretamente, se implementa como una función en Haskell:

```
tokenize :: String → [Token]
```

Los tokens se representan como un tipo de datos enumerado que, en el caso de los literales, viene acompañado además del dato correspondiente. Además, un token incluye información sobre la posición inicial y la posición final en la que se encuentra el lexema dentro del archivo fuente. Esta información es relevante tanto para posibilitar el reporte de errores (indicando número de fila y columna

del error) como para la implementación de las reglas de *layout*, que son sensibles a la identación.

El analizador léxico se encuentra en el módulo Lexer de la implementación de Ñuflo. No presenta demasiadas dificultades desde el punto de vista de la implementación, salvo por las reglas de *layout*.

Implementación de las reglas de *layout*.

El algoritmo que se utiliza para la resolución de las reglas de *layout* en el lenguaje es un algoritmo *ad hoc* que utiliza como referencia la documentación del reporte de Haskell 98 [**?**]. Más específicamente, se sigue el algoritmo detallado en la sección 9.3 de [**?**], que implementa las reglas de *layout* para Haskell, con algunas adaptaciones.

El efecto de resolver las reglas de *layout* resulta en la adición de llaves, es decir, los tokens `LBRACE` (`{`) y `RBRACE` (`}`), así como de puntos y coma, es decir el token `SEMICOLON` (`;`) en los lugares donde las reglas de *layout* lo indiquen.

Se dice que el programa resultante de aplicar este algoritmo se vuelve "insensible a la identación", ya que la información de dónde comienza y dónde termina cada bloque está determinada únicamente por los tipos de los tokens y no por sus posiciones.

El algoritmo que implementa las reglas de *layout* se encuentra en el módulo Lexer.Layout de la implementación de Ñuflo.

2. Sintaxis

La sintaxis de Ñuflo está influenciada por la sintaxis de lenguajes como Haskell y Agda. En particular, Ñuflo toma prestada de Agda la notación para definir operadores prefijos, sufijos, infijos y, más en general, operadores *multifijos*, especificando su asociatividad y su nivel de precedencia, lo cual le proporciona al programador una mayor versatilidad para crear nuevas operaciones.

Los operadores multifijos se representan por medio de un identificador que incluye una o varias veces el caracter guión bajo (`_`). Por ejemplo, si se declara el operador `if_then_else_` con nivel de precedencia 10, y el operador `_*_` con nivel de precedencia 20 y asociatividad a derecha, la siguiente expresión:

```
if a then b * c * d else e
```

es equivalente a esta otra:

```
if_then_else_ a (_*_ b (_*_ c d)) e
```

2.1. Declaraciones

Un programa en Ñuflo se compone de varios módulos. Cada módulo incluye su declaración (ej. `module Data.List where`) y una lista de módulos que se

importan (ej. `import Data.Char`). A continuación, sigue una *lista de declaraciones*. Cada declaración puede ser de alguna de las siguientes formas:

1. **Declaración de un operador**:

   ```
   infixl 20 _+_
   ```

   Conformada por el tipo de asociatividad del operador: ya sea si asocia a derecha (`infixr`), a izquierda (`infixl`) o si es no asociativo (`infix`). En la declaración se incluye además un número natural que indica el nivel de precedencia y el nombre del operador, que debe incluir al menos un guión bajo (`_`).

2. **Declaración de un tipo de datos**:

   ```
   data Tree a where
     Nil : Tree a
     Bin : Tree a → a → Tree a → Tree a
   ```

   Similar a la notación de Haskell, un tipo de datos se declara a través de un constructor de tipo con sus parámetros formales, seguido de un bloque de signaturas de tipo para cada uno de los constructores del tipo.

3. **Declaración de un sinónimo de tipos**:

   ```
   type TreeList a = Tree (List a)
   ```

   Una declaración de tipo puede verse como un *alias*, donde a partir de tipos ya definidos se define un nuevo tipo de datos, con idéntica notación a la usada en Haskell para el mismo propósito.

4. **Declaración de una clase**:

   ```
   class Monad m where
     return : a → m a
     _>>=_  : m a → (a → m b) → m b
   ```

   La declaración de una clase está conformada por el nombre de la clase, seguido por la variable de tipo ligada en las definiciones de los métodos abstractos. A continuación, se incluye un bloque de signaturas que determinan los tipos de los métodos de la clase. La sintaxis es similar a la de las declaraciones de *typeclasses* en Haskell.

5. **Declaración de una instancia**:

```
instance Monad List where
  return x = x : []
  xs >>= f = concatMap f xs
```

La declaración de una instancia está conformada por el nombre de la clase acompañada del tipo que la implementa. El tipo debe ser siempre de la forma `(C a1 ... an)` donde `C` es el nombre de un constructor, y `a1`, ..., `an` son variables de tipo. Además se debe dar una definición para cada uno de los *métodos* requeridos por esa clase.

La notación es similar a la de las declaraciones de instancia Haskell, con la diferencia de cómo se expresan las instancias en las cuales se requieren a su vez restricciones de clase para las variables de tipo. Por ejemplo, una lista de elementos implementa la clase `Show` si a su vez el tipo de sus elementos implementa la clase `Show`:

```
instance Show (List a) {Show a} where
  show []       = "[]"
  show (x : xs) = show x ++ " : " ++ show xs
```

Si hay varias restricciones de clase se separan con punto y coma (`;`), por ejemplo `{Show a; Ord a; Eq b}`.

6. **Declaración de un valor o una función/relación**:

```
fix f = f (fix f)
```

Los valores (que representan datos, o funciones/relaciones) se definen a través de ecuaciones. Cada ecuación contribuye a darle valor a la variable que se encuentra en la *cabeza* de la expresión del lado izquierdo. (La cabeza de una expresión `e` es la expresión `e'` tal que `e` se escribe como una aplicación `e' x1 ... xn`, y además `e'` no es una aplicación).

Además, cada ecuación puede venir seguida de una cláusula `where`:

```
partes []       = [] : []
partes (x : xs) = p ++ map (_:_ x) p
  where p = partes xs
```

7. **Declaración de una signatura de tipo**:

```
elem : a -> List a -> Bool    {Eq a}
```

Las declaraciones de valores pueden venir acompañadas de signaturas de tipo. Una signatura de tipo está conformada por un identificador, seguido del símbolo `:`, seguido de una expresión que representa el tipo. A continuación, opcionalmente, se incluyen restricciones de clase para las variables tipo, encerradas entre llaves. Como ya fue mencionado, si hay varias restricciones de clase, se separan con punto y coma (`;`), por ejemplo `{Show a; Ord a; Eq b}`.

2.2. Expresiones

Por otro lado, los tipos, así como el lado izquierdo y el lado derecho de las ecuaciones son **expresiones**. Las expresiones pueden ser, inductivamente, un *átomo* o una aplicación de una expresión a otra, teniendo en cuenta que la aplicación es asociativa a izquierda, de modo que `f x y z = ((f x) y) z`. Se respetan las asociatividades y niveles de precedencia declarados para los operadores multifijos, y se pueden utilizar paréntesis para agrupar expresiones (desestimando las declaraciones de asociatividad y precedencia). Los átomos son los siguientes:

1. **Nombre calificado:** un átomo puede ser un *nombre calificado* (también llamado `QName`), es decir un identificador, posiblemente prefijado por el nombre del módulo en el que está declarado. Por ejemplo, son nombres calificados:

   ```
   omega   if_then_else_   _*_   ,   Data.List.filter   Monad._>>=_
   ```

2. **Constantes:** las constantes pueden ser literales enteros positivos (ej. `42`), literales de caracter (ej. `'a'`), y strings (ej. `"el perro feo"`).

3. **Declaraciones locales (`let`):** la sintaxis de una declaración local es de la forma

   ```
   let <declaraciones> in <expr>
   ```

   donde `<declaraciones>` es una secuencia de declaraciones de tipo o de valor (ecuaciones). Por ejemplo:

   ```
   partes (x : xs) = let p  = partes xs
                         p' = map (_:_ x) p
                     in p ++ p'
   ```

   Recordar que `let` es una palabra clave *offside*, por lo cual todas las declaraciones deben alinearse en la misma columna.

4. **Abstracciones anónimas (\ ó λ):** una abstracción anónima o "lambda" se acompaña de una secuencia de *expresiones* que representan los parámetros formales de la abstracción, seguida de una flecha `->`, y seguida de una expresión que representa el cuerpo de la abstraccion. Por ejemplo, la composición se puede denotar a través de la siguiente expresión:

   ```
   \ f g x -> f (g x)
   ```

   Observar que los parámetros formales pueden ser *patrones* arbitrariamente complejos. Por ejemplo, la función/relación que recibe una lista y devuelve su cola (o falla si la lista es vacía) se puede escribir como:

   ```
   \ (_ : xs) -> xs
   ```

   Se puede presentar una ambigüedad con respecto a si una variable que aparece dentro de un patrón se trata de una variable ligada por esa abstracción, o una variable ligada por alguna construcción más externa. Por ejemplo, si el nombre `Nil` no se encuentra declarado en el contexto local, en la abstracción `(\ Nil -> True)` el identificador `Nil` se interpreta como el nombre del parámetro formal, y así dicha abstracción representa la función que constantemente devuelve la constante `True`. En cambio, si el nombre `Nil` es un constructor ya declarado, el identificador `Nil` se interpreta como una referencia a ese constructor, y así dicha abstracción representa la función que devuelve `True` cuando su parámetro coincide con `Nil` pero falla en caso contrario.

   Para forzar a que un identificador dentro de un patrón se interprete como un parámetro formal (y no como una referencia a un nombre externo), se lo puede preceder de un punto (`.`). Por ejemplo, `(\ (.Nil) -> Nil)` representa la función identidad, independientemente de si `Nil` ya se encontraba localmente declarado.

5. **Alternativa por casos (`case`):** la alternativa por casos se escribe con la palabra clave `case`, seguida de una expresión a analizar, seguida de la palabra clave `of`, y seguida de una secuencia de ramas.

   Cada rama representa el comportamiento ante una posible coincidencia, formada por un *patrón*, seguido de una flecha (`->`), seguida de la expresión a evaluar en caso de que la expresión analizada coincida con ese patrón. Por ejemplo:

   ```
   last lista = case lista of
                    (x : []) → x
                    (_ : xs) → last xs
   ```

Esta operación está emparentada con la sentencia "`switch`" de otros lenguajes, y sigue la notación del *case* de Haskell. Recordar que `of` es una palabra clave *offside*, por lo cual las ramas deben estar alineadas sobre la misma columna.

Además, a diferencia de lo que ocurre en Haskell, las ramas **no** son mutuamente excluyentes, de manera tal que si una expresión coincide con varios patrones, se devuelven los valores de todas las ramas que coinciden, no determinísticamente.

Al igual que en el caso de las abstracciones anónimas, se puede forzar a que una variable sea ligada localmente en esa rama prefijándola por un punto (`.`).

6. **Introducción de variable fresca (`fresh`):** la operación de introducción de una variable fresca se escribe con la palabra clave `fresh`, seguida de una secuencia de identificadores, seguida de la palabra clave `in`, y seguida, por último, de una expresión que denota el cuerpo de la operación. Por ejemplo:

```
rotaciones lista = fresh x y in (lista ~ x ++ y) & (y ++ x)
```

Esta operación esta influenciada por la necesidad de instanciación de variables frescas del paradigma lógico, aunque difiere de la sintaxis usada en Prolog.

2.3. Gramática formal

Formalmente, la gramática de Ñuflo se define de la siguiente manera usando notación BNF:

⟨*program*⟩ ⟶ `MODULE` ⟨*qname*⟩ ⟨*moduleExports*⟩ `WHERE LBRACE` ⟨*moduleImports*⟩ ⟨*declaration**⟩ `RBRACE`

⟨*qname*⟩ ⟶ `ID`
| `ID DOT` ⟨*qname*⟩

⟨*moduleExports*⟩ ⟶ ϵ
| `LPAREN` ⟨*id**⟩ `RPAREN`

⟨*id**⟩ ⟶ ϵ
| `ID SEMICOLON` ⟨*id+*⟩

⟨*id+*⟩ ⟶ `ID`
| `ID SEMICOLON` ⟨*id+*⟩

⟨*moduleImports*⟩ ⟶ ϵ
| `IMPORT` ⟨*qname*⟩ ⟨*optionalRenamings*⟩ ⟨*moduleImports*⟩

⟨*optionalRenamings*⟩ ⟶ ϵ
| LPAREN ⟨*renaming*⟩* RPAREN

⟨*renaming+*⟩ ⟶ ⟨*renaming*⟩
| ⟨*renaming*⟩ SEMICOLON ⟨*renaming+*⟩

⟨*renaming**⟩ ⟶ ϵ
| ⟨*renaming*⟩ SEMICOLON ⟨*renaming+*⟩

⟨*renaming*⟩ ⟶ ID
| ID AS ID

⟨*declaration**⟩ ⟶ ϵ
| ⟨*declaration+*⟩

⟨*declaration+*⟩ ⟶ ⟨*declaration*⟩
| ⟨*declaration*⟩ SEMICOLON ⟨*declaration+*⟩

⟨*declaration*⟩ ⟶ IMPORT ⟨*qname*⟩ ⟨*moduleImports*⟩
| IMPORT ⟨*qname*⟩ ⟨*moduleImports*⟩ AS ID
| ⟨*fixity*⟩ INT ID
| DATA ⟨*expr*⟩ WHERE LBRACE ⟨*typeSignature**⟩ RBRACE
| TYPE ⟨*expr*⟩ EQ ⟨*expr*⟩
| CLASS ⟨*classname*⟩ ⟨*qname*⟩ ⟨*optionalConstraints*⟩ WHERE LBRACE ⟨*typeSignature**⟩ RBRACE
| INSTANCE ⟨*classname*⟩ ⟨*expr*⟩ ⟨*optionalConstraints*⟩ WHERE LBRACE ⟨*declaration*⟩ RBRACE
| ⟨*typeSignature*⟩
| ⟨*equation*⟩

⟨*fixity*⟩ ⟶ INFIXL
| INFIXR
| INFIX

⟨*typeSignature**⟩ ⟶ ϵ
| ⟨*typeSignature+*⟩

⟨*typeSignature+*⟩ ⟶ ⟨*typeSignature*⟩
| ⟨*typeSignature*⟩ SEMICOLON ⟨*typeSignature+*⟩

⟨*typeSignature*⟩ ⟶ ID COLON ⟨*expr*⟩ ⟨*optionalConstraints*⟩

⟨*equation**⟩ ⟶ ϵ
| ⟨*equation+*⟩

⟨*equation+*⟩ ⟶ ⟨*equation*⟩
| ⟨*equation*⟩ SEMICOLON ⟨*equation+*⟩

⟨*equation*⟩ ⟶ ⟨*expr*⟩ EQ ⟨*expr*⟩
| ⟨*expr*⟩ EQ ⟨*expr*⟩ WHERE LBRACE ⟨*declaration**⟩ RBRACE

$$
\begin{array}{lcl}
\langle optionalConstraints \rangle & \longrightarrow & \epsilon \\
& | & \texttt{LBRACE}\ \langle constraint* \rangle\ \texttt{RBRACE}
\end{array}
$$

$$
\begin{array}{lcl}
\langle constraint* \rangle & \longrightarrow & \epsilon \\
& | & \langle constraint+ \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle constraint+ \rangle & \longrightarrow & \langle constraint \rangle \\
& | & \langle constraint \rangle\ \texttt{SEMICOLON}\ \langle constraint+ \rangle
\end{array}
$$

$$
\langle constraint \rangle \longrightarrow \langle classname \rangle\ \langle qname \rangle
$$

$$
\langle classname \rangle \longrightarrow \langle qname \rangle
$$

$$
\begin{array}{lcl}
\langle expr \rangle & \longrightarrow & \langle atom \rangle \\
& | & \langle atom \rangle\ \langle expr \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle atom \rangle & \longrightarrow & \langle qname \rangle \\
& | & \texttt{DOT ID} \\
& | & \texttt{INT} \\
& | & \texttt{CHAR} \\
& | & \texttt{STRING} \\
& | & \texttt{LPAREN}\ \langle expr \rangle\ \texttt{RPAREN} \\
& | & \texttt{LET LBRACE}\ \langle declaration* \rangle\ \texttt{RBRACE IN}\ \langle expr \rangle \\
& | & \texttt{LAMBDA}\ \langle id* \rangle\ \texttt{ARROW}\ \langle expr \rangle \\
& | & \texttt{CASE}\ \langle expr \rangle\ \texttt{OF LBRACE}\ \langle branch* \rangle\ \texttt{RBRACE} \\
& | & \texttt{FRESH LBRACE}\ \langle idseq* \rangle\ \texttt{RBRACE ID}\ \langle expr \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle idseq* \rangle & \longrightarrow & \epsilon \\
& | & \langle id* \rangle\ \texttt{SEMICOLON}\ \langle idseq+ \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle idseq+ \rangle & \longrightarrow & \langle id* \rangle \\
& | & \langle id* \rangle\ \texttt{SEMICOLON}\ \langle idseq+ \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle id* \rangle & \longrightarrow & \epsilon \\
& | & \texttt{ID}\ \langle id* \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle branch* \rangle & \longrightarrow & \epsilon \\
& | & \langle branch+ \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle branch+ \rangle & \longrightarrow & \langle branch \rangle \\
& | & \langle branch \rangle\ \texttt{SEMICOLON}\ \langle branch+ \rangle
\end{array}
$$

$$
\langle branch \rangle \longrightarrow \langle atom \rangle\ \texttt{ARROW}\ \langle expr \rangle
$$

2.4. Analizador sintáctico

En esta sección se detalla el funcionamiento del analizador sintáctico de Ñuflo. El *parser* de Ñuflo está implementado manualmente, utilizando un método de **descenso recursivo** *ad hoc* (sin recurrir a herramientas como generadores de parsers).

El analizador sintáctico puede verse como una función del siguiente tipo:

```
parse :: [Token] -> AST
```

que recibe un programa ya segmentado en lexemas, es decir una lista de *tokens*, y retorna el árbol de sintaxis abstracta (AST) correspondiente a dicho programa.

Los aspectos clave de la implementación del *parser* son el detalle del análisis de expresiones que involucran operadores multifijos, así como la estructura del *árbol de sintaxis abstracta*.

El *parser*, al igual que el *lexer*, tiene dos tareas. La primera de ellas es detectar errores de sintaxis en la entrada. Por otra parte, en caso de que el programa sea gramaticalmente correcto, se debe construir una representación interna arbórea que representa abstractamente la estructura de la expresión. Por ejemplo, en el AST no se incluyen ciertos elementos como puntuación no esencial o delimitadores.

2.4.1. Estructura del AST

Un programa Ñuflo se representa como un valor de tipo `Program` en Haskell. Un nodo `Program` es a su vez una lista de nodos `Declaration`. Los nodos de tipo `Declaration` representan las declaraciones que pueden encontrarse en un programa. Pueden tener las siguientes formas, con sus respectivos campos:

1. `DataDeclaration`: declaración de un tipo de datos inductivo.
   - `dataType :: Expr` -- Tipo de datos.
   - `dataConstructors :: [Signature]` -- Lista de constructores.
2. `TypeDeclaration`: declaración de un sinónimo de tipos.
   - `typeType :: Expr` -- Tipo de datos.
   - `typeValue :: Expr` -- Definición.
3. `TypeSignature:` declaración de tipo para un identificador.
   - `typeSignature :: Signature` -- Declaración.
4. `ValueDeclaration:` declaración de un valor (ecuación).
   - `declEquation :: Equation` -- Ecuación.

5. `ClassDeclaration`: declaración de una clase.

   - `className :: QName` -- Nombre de la clase.
   - `classTypeName :: QName` -- Parámetro formal.
   - `classMethods :: [Signature]` -- Signaturas de métodos.

6. `InstanceDeclaration`: declaración de una instancia.

   - `instanceClassName :: QName` -- Nombre de la clase.
   - `instanceType :: Expr` -- Tipo instanciado.
   - `instanceConstraints :: [Constraint]` -- Restricciones de clase.
   - `instanceMethods :: [Equation]` -- Definiciones de métodos.

Los nodos de tipo `Signature` representan la declaración del tipo de un nombre (que puede ser una función/relación, un constructor, un método, etc.). Tiene los siguientes campos:

- `signatureName :: QName` -- Nombre declarado.
- `signatureType :: Expr` -- Tipo.
- `signatureConstraints :: [Constraint]` -- Restricciones de clase.

Los nodos de tipo `Equation` representan las ecuaciones o declaraciones de valores. Tiene los siguientes campos:

- `equationLHS :: Expr` -- Lado izquierdo de la ecuación.
- `equationRHS :: Expr` -- Lado derecho de la ecuación.

Los nodos de tipo `Constraint` representan restricciones de clase asociadas a una variable de tipo. Tiene los siguientes campos:

- `constraintClassName :: QName` -- Nombre de la clase.
- `constraintTypeName :: QName` -- Variable de tipo.

Los nodos de tipo `CaseBranch` representan las ramas de una alternativa por casos (`case`). Tiene los siguientes campos:

- `caseBranchPattern :: Expr` -- Patrón.
- `caseBranchResult :: Expr` -- Cuerpo de la rama.

Los nodos de tipo `Expr` representan expresiones. Pueden tener las siguientes formas:

1. `(EVar x)`: representa una variable, donde `x :: QName` es su nombre.

2. `(EUnboundVar x)`: representa una variable de nombre `x :: QName` que está explícitamente ligada por el patrón actual, lo que se indica con un punto (`.`) en el código fuente,

3. `(EInt n)`: representa una constante entera de valor `n :: Integer`.

4. `(EChar c)`: representa una constante de caracter de valor `c :: Char`.

5. `(EApp e1 e2)`: representa la aplicación de la función `e1 :: Expr` al argumento `e2 :: Expr`.

6. `(ELambda e1 e2)`: representa una abstracción donde la expresión `e1 :: Expr` representa el patrón y la expresión `e2 :: Expr` representa el cuerpo.

7. `(ELet decls e)`: representa una declaración local (`let`), conformada por una *lista* de declaraciones `decls :: [Declaration]` y un cuerpo `e :: Expr`.

8. `(ECase e branches)`: representa una alternativa por casos (`case`) conformada por una expresión `e :: Expr` a analizar y una *lista* de ramas `branches :: [CaseBranch]`.

9. `(EFresh x e)`: representa la introducción de la variable fresca `x :: QName` ligada en el cuerpo `e :: Expr`.

10. `(EPlaceholder id)`: este nodo del AST no corresponde a una construcción que pueda escribir el usuario. Es utilizado por el inferidor de tipos, y representa un hueco a rellenar con la implementación de los métodos que imponen las restricciones de clase. Esto es, en caso de tener una función polimórfica que depende de una variable de tipo que se incluye con restricciones de clase, la función depende de la instancia de la clase que satisface dicha restricción. El inferidor de tipos introduce explícitamente estos "huecos" o "*placeholders*", y los completa en diferido a la hora de resolver las restricciones de clase. Cada nodo `EPlaceholder` viene acompañado de un `id` de tipo `PlaceholderId`, que no es más que un número que sirve para identificar internamente a dicho hueco.

Los nodos del AST describen abstractamente las construcciones del lenguaje. Además, cada nodo del AST incluye información sobre la posición de cada subexpresión en el código fuente del programa. Esto es relevante para el reporte de errores.

2.4.2. Análisis de operadores multifijos

Como se mencionó anteriormente, el analizador sintáctico respeta las declaraciones de asociatividad y precedencia de cada operador multifijo. El análisis

sintáctico de expresiones que incluyen operadores multifijos se resuelve con un algoritmo *ad hoc* que se describe a continuación.

Debemos tener en cuenta que en cualquier expresión pueden aparecer combinaciones arbitrarias de aplicaciones de operadores multifijos con diversas asociatividades y niveles de precedencia. Los operadores pueden aparecer dentro o fuera de paréntesis, que deben respetarse siguiendo la convención usual de paréntesis. Por esto, el analizador sintáctico debe utilizar una *tabla de precedencia* que reúne la información sobre la asociatividad y los niveles de precedencia de cada uno de los operadores. Esto no es tan sencillo porque en Ñuflo los operadores pueden estar declarados por el usuario. Esto implica que el *parser*, previo a realizar el análisis del programa, debe recolectar todas las definiciones de operadores multifijos, construyendo la tabla de precedencia y almacenando el tipo de asociatividad y el nivel de precedencia de cada operador. Además, se mantiene registro en la tabla acerca de cuáles nombres conforman *partes* de algún operador. Por ejemplo, si el usuario declara el siguiente operador:

```
infix 20 if_then_else_
```

los identificadores `if`, `then` y `else` se registran como *partes* de algún operador. Notar que un mismo identificador puede componer varios operadores diferentes. Por ejemplo, si el operador `if_then_else` ya se encuentra definido, puede definirse otro operador `unless_then_`, a pesar de que el identificador `then` sea parte de ambos operadores.

El algoritmo para analizar sintácticamente una expresión es un algoritmo recursivo que tiene como parámetro un nivel ℓ y analiza sintácticamente una expresión de nivel ℓ. Las expresiones de nivel ℓ pueden incluir operadores que tengan nivel de precedencia ℓ o mayor que ℓ *no rodeados de paréntesis*. Además, pueden incluir cualquier otro operador *rodeado de paréntesis*, inclusive operadores con nivel de precedencia estrictamente menor que ℓ. Supongamos, por ejemplo, que ya se encuentra construida la siguiente tabla de precedencia:

Operador	**Precedencia**	**Asociatividad**
`if_then_else_`	20	no asociativo
$\_\oplus\_$	30	asociativo a derecha

Al analizar la expresión:

`if a then e` $\oplus$ `f else g`

el analizador sintáctico procede del siguiente modo:

1. Comienza el análisis de una expresión de nivel 20, que es el menor nivel de precedencia presente en la tabla.

2. Al encontrar el identificador `if`, que corresponde a una *parte* de un operador de ese nivel, consume el operador y registra que el prefijo consumido en el nivel actual es de la forma "`if...`".

3. Como el prefijo "`if...`" no es la aplicación de un operador completo, procede a analizar recursivamente una expresión de nivel 30, que es el siguiente nivel de precedencia en la tabla. Como resultado se consume el identificador `a` de la entrada produciendo como resultado un AST X, y el prefijo consumido en el nivel actual pasa a ser de la forma "`if_...`"

4. Como el prefijo "`if_...`" no es la aplicación de un operador completo y el identificador `then` corresponde a una posible continuación del prefijo actual, se consume dicho identificador y el prefijo consumido en el nivel actual pasa a ser "`if_then...`".

5. Como el prefijo "`if_then...`" no es la aplicación de un operador completo procede a analizar recursivamente una expresión de nivel 30, que es el siguiente nivel de precedencia en la tabla. Como resultado se consume la subexpresión `(if b then c else d)` $\oplus$ `e` $\oplus$ `f` produciendo como resultado un AST Y, y el prefijo consumido en el nivel actual pasa a ser de la forma "`if_then_...`".

6. Como el prefijo "`if_then_...`" no es la aplicación de un operador completo y el identificador `else` corresponde a una posible continuación del prefijo actual, se consume dicho identificador y el prefijo consumido en el nivel actual pasa a ser "`if_then_else...`".

7. Como el prefijo "`if_then_else...`" no es la aplicación de un operador completo procede a analizar recursivamente una expresión de nivel 30, que es el siguiente nivel de precedencia en la tabla. Como resultado se consume el identificador `g` produciendo como resultado un AST Z, y el prefijo consumido en el nivel actual pasa a ser de la forma "`if_then_else_...`".

8. Finalmente, el prefijo "`if_then_else_...`" sí corresponde a un operador completamente aplicado. El AST que se produce es el siguiente:

   `EApp (EApp (EApp (EVar "if_then_else_")` X`)` Y`)` Z

   que corresponde a la aplicación de la variable `"if_then_else_"` a los argumentos X, Y y Z.

El mecanismo general de análisis sintáctico de operadores multifijos sigue estas mismas ideas, con adaptaciones para permitir operadores asociativos a izquierda y a derecha, y para reportar errores de sintaxis.

En el caso particular del lenguaje Ñuflo, notar que las operaciones de unificación, alternativa no deterministica y secuencia se expresan como operadores (`_~_`, `_|_`, y `_&_` respectivamente). Ninguno de estos operadores está presente explícitamente en la definición de la gramática. Son operadores primitivos que se incluyen por defecto en la tabla de precedencia del parser.

3. Sistema de módulos

En la sección anterior se mencionó que un programa consta de una secuencia de declaraciones de módulos. Cada módulo puede, dentro de su definición, especificar qué nombres serán exportados. Es decir, qué nombres definidos en este módulo podrán ser importados por otros módulos. La forma de declarar los nombres que importa un módulo es a través de la declaración `import`.

Como se puede observar, en la gramática existen dos tipos de declaración `import`. Esto se debe a que el lenguaje provee mecanismos de *renombre* para evitar conflictos entre un nombre que se importa a través de una declaración `import` y algún nombre local. El sistema de módulos se describirá informalmente con algunos ejemplos.

Si quisiéramos definir un nuevo módulo C que se encuentra dentro de otro módulo B y al mismo tiempo B se encuentra dentro de un módulo A, entonces la definición de C tendrá la forma:

```
module A.B.C where
```

Los nombres de los módulos deben respetar la estructura de directorios. Por ejemplo, el código correspondiente al módulo `A.B.C` se encontrará dentro del archivo "`A/B/C.nu`".

Observar que, de esta manera, el módulo definido esta exportando todos los nombres locales que define. En caso de querer evitar esto, el sistema de módulos permite especificar qué nombres se desea exportar. Por ejemplo, si sólo se desea exportar los nombres `Tree`, `EmptyT`, `NodeT` y `height`, se puede hacer de la siguiente manera:

```
module A.B.C (Tree; EmptyT; NodeT; height) where
```

Supongamos que se define ahora un módulo `D` y que es necesario utilizar los nombres `Tree`, `EmptyT`, `NodeT` y `height` en `D`. La forma de explícitar esta dependencia es mediante una declaración `import` como la siguiente:

```
import A.B.C
```

Haciendo esto, tendremos acceso a todos los nombres exportados por el módulo `A.B.C` dentro de `D`. Por otro lado, se puede importar sólo un subconjunto de los nombres exportados por un módulo. Por ejemplo, si sólo es necesario utilizar el nombre `EmptyT` del modulo `A.B.C`, podemos hacerlo del siguiente modo:

```
import A.B.C (EmptyT)
```

Con esta declaración, el identificador `EmptyT` "desnudo" se refiere al nombre `EmptyT` declarado en el módulo `A.B.C`. También se puede hacer referencia a

cualquier otro nombre exportados por el módulo `A.B.C` pero en ese caso será necesario *calificarlo* con el prefijo `A.B.C`, escribiendo por ejemplo `A.B.C.Tree`. Observar en particular que la declaración `import A.B.C ()` importa todos los nombres del módulo `A.B.C` pero siempre calificados.

Una forma de abreviar el prefijo con el que se califican los nombres es mediante un *alias*. Para darle un alias local el módulo `A.B.C` importado basta con utilizar la palabra clave *as*. Por ejemplo, la siguiente declaración:

```
import A.B.C () as Z
```

permite usar todos los nombres exportados por el módulo `A.B.C` siempre que se los califique con el prefijo `Z`, por ejemplo, `Z.EmptyT`.

Por último, para evitar posibles conflictos de nombres, es posible importar nombres explícitamente de un módulo pero renombrándolos. Por ejemplo:

```
import A.B.C (EmptyT as Nil; NodeT as Bin)
```

En conclusión, el sistema de módulos permite definir e importar módulos, especificando qué nombres se exportan e importan, en conjunto con un mecanismo de alias/renombre.

4. Sistema de tipos

Un sistema de tipos es un conjunto de reglas que se encargan de asociar un atributo o propiedad denominada *tipo* a cada fragmento del programa escrito en un lenguaje de programación. El propósito principal del sistema de tipos, desde nuestro punto de vista, es asegurar que los programas estén construidos de manera "coherente", reportando un error de tipos en el caso contrario.

Muchos lenguajes, como C y Java por ejemplo, requieren anotaciones de tipos explícitas en el programa. Es decir que el programador debe declarar manualmente los tipos de todos los identificadores. Otros lenguajes, como OCaml y Haskell, usan un algoritmo de *inferencia de tipos*. Esto significa que el compilador es capaz de determinar los tipos de las expresiones analizando la estructura del programa. Este análisis se hace en forma estática, es decir en tiempo de compilación. Por ejemplo, si se declara la siguiente función en Haskell:

```
f x y = if y then x + 1 else 0
```

es posible inferir que el tipo de `f` es `Int -> Bool -> Int`.

En esta sección se describe informalmente la implementación de un inferidor de tipos del lenguaje de programación Ñuflo. El sistema de tipos de Ñuflo sigue fuertemente la línea de los sistemas de la familia de Damas–Hindley–Milner. Esta familia de sistemas de tipos se caracteriza por contar con *polimorfismo*

paramétrico, es decir, funciones y tipos de datos "genéricos" que operan con valores independientemente de su tipo. Por ejemplo, consideremos la siguiente definición de la función `map`:

```
map f []       = []
map f (x : xs) = f x : map f xs
```

Es posible darle muchos tipos distintos a `map`. Por ejemplo, los dos tipos siguientes son tipos posibles para la función `map`:

```
(Int → Int) → List Int → List Int
(Bool → List Int) → List Bool → List (List Int)
```

El algoritmo de inferencia de tipos le otorga a `map` el tipo más general posible (conocido como su "tipo principal"), que es de la forma:

```
(a → b) → List a → List b
```

donde `a` y `b` son variables de tipo que pueden instanciarse en tipos arbitrarios cada una de las veces que se utiliza la función `map`. Observar que los dos posibles tipos de `map` mencionados más arriba son instancias de su tipo más general.

Además, el sistema de tipos de Ñuflo, igual que el sistema de tipos de Haskell, está extendido con *typeclasses* (o "clases de tipos"), que agregan soporte para hacer *polimorfismo ad hoc*. Una *typeclass*, declarada con la palabra clave `class`, representa una interfaz que depende de un tipo `a`, especificada por un conjunto de nombres de métodos con sus respectivos tipos. Distintos constructores de tipos pueden implementar esa interfaz, lo que se expresa con la palabra clave `instance`.

Por ejemplo, el siguiente fragmento de código declara una clase `Eq` que corresponde a los tipos que implementan un operador de igualdad `_==_`, y define la instancia de la clase `Eq` para el tipo de los *bits*.

```
class Eq a where
  _==_ : a → a → Bool

data Bit where
  Zero : Bit
  One  : Bit

instance Eq Bit where
  (Zero == Zero) = True
  (Zero == One)  = False
  (One  == Zero) = False
  (One  == One)  = True
```

4.1. Inferencia de *kinds*

Algunos sistemas de tipos definen la noción de *kind*, que corresponde a los "tipos de las expresiones de tipos", y sirven para garantizar la validez de las expresiones de tipos. Por ejemplo, si `Int` es un tipo y `List` es un constructor de tipos unario, entonces `(List Int)` y `(List (List Int))` serán tipos bien formados, en tanto que `(List List)` será una expresión de tipos mal formada, a la que no se le puede asignar un *kind*.

Un *kind* puede tener alguna de las siguientes formas:

- El *kind* de los tipos básicos se nota `*`.
- Si $\texttt{k}_1$ y $\texttt{k}_2$ son *kinds*, $\texttt{k}_1 \to \texttt{k}_2$ será el *kind* de los constructores de tipos que reciben como parámetro un *tipo* de *kind* $\texttt{k}_1$ y devuelven un tipo de *kind* $\texttt{k}_2$.

Por ejemplo, `Int` es un tipo básico de *kind* $*$, mientras que `List` es un constructor de tipos de *kind* $* \to *$.

El proceso de inferencia de *kinds* es una etapa preliminar del sistema de tipos, que verifica la buena formación de las expresiones de tipo (de forma análoga al proceso de inferencia de tipos, que se encarga de comprobar la buena formación de las expresiones).

A diferencia de los tipos, los *kinds* están completamente implícitos en los programas y no forman parte visible del lenguaje. Sólo existen durante el proceso de compilación de un programa. El mecanismo de inferencia de *kinds* debe determinar:

- El *kind* de cada constructor de tipo y de sus parámetros. Por ejemplo, en la definición:

```
data Tree a b where
  Nil  : Tree a b
  Node : a → b → Tree a b → Tree a b → Tree a b
```

  los parámetros `a` y `b` tienen kind $*$ y el constructor de tipos `Tree` tiene kind $* \to * \to *$.

- El *kind* de cada sinónimo de tipo que se defina y de sus parámetros. Por ejemplo, en la definición:

```
type LT f = List (f Tree)
```

  el parámetro `f` tiene kind $(* \to * \to *) \to *$ y el sinónimo de tipos `LT` tiene kind $((* \to * \to *) \to *) \to *$.

- El *kind* del parámetro de cada *typeclass*. Por ejemplo, en la declaración:

```
class Monad m where
  return : a → m a
  _>>=_  : (a → m b) → m a → m b
```

  el parámetro `m` tiene *kind* $* \to *$.

Es posible que algunas partes de un *kind* inferido no estén completamente determinadas por las correspondientes definiciones, en tal caso se asume el valor por defecto $*$. Por ejemplo, podríamos asumir un *kind* arbitrario para el parámetro a en la siguiente definición:

```
data App f a where
  A : f a → App f a
```

dándole el *kind* $(\mathtt{k} \to *) \to \mathtt{k} \to *$ y $\mathtt{k} \to *$ al constructor de tipos `App`. Utilizando la asignación por defecto $\mathtt{k} = *$ su kind resulta ser $(* \to *) \to * \to *$.

El proceso de inferencia de *kinds* se basa en un algoritmo de unificación de *kinds*. Por ejemplo, si sabemos que el constructor de tipos `A` tiene *kind* $(\mathtt{k}_1 \to *)$ y el constructor de tipos `B` tiene *kind* $(* \to *)$, al encontrar la expresión de tipos `A B`, se debe unificar $\mathtt{k}_1$ con $(* \to *)$. Así el *kind* del tipo `A` será $((* \to *) \to *)$. Si el proceso de unificación falla, se reporta un error de *kinds* en el programa.

4.2. Inferencia de tipos

El problema de inferencia de tipos consiste en determinar, dado un término t, si existen un contexto de tipado Γ y un tipo A tales que el juicio de tipado $\Gamma \vdash t : A$ es derivable (en algún sistema de tipos prefijado).

En algunos sistemas de tipos, el problema de inferencia es decidible. Incluso si el problema de inferencia de tipos es indecidible, muchas veces es posible resolverlo para un subconjunto representativo o interesante de programas.

Por ejemplo, Haskell '98 implementa un sistema de tipos al estilo de Hindley–Milner [**?**], que es una restricción de System F en el que la inferencia de tipos resulta decidible.

El sistema de tipos de Ñuflo es similar al sistema de Haskell '98. Nos limitaremos a describir *informalmente* la representación de los tipos y el mecanismo de inferencia. Distinguimos entre: *tipos* a secas, *tipos restringidos* y *esquemas* de

tipos, que se definen como sigue:

Tipos (a secas)	τ	::=	α	variable de tipos
		|	$\tau\,\tau$	aplicación
		|	`?X`	metavariable de tipos
Tipos restringidos	Q	::=	$\tau\ \{C_1\alpha_1; \ldots; C_n\alpha_n\}$	
Esquemas	E	::=	$\forall\ \alpha_1 \ldots \alpha_n\ .\ Q$	

Los *tipos* a secas designan un tipo particular. Un tipo puede ser una variable de tipos, que incluye tanto nombres de constructores de tipos (ej. `Int`, `Bool`, `List`, `_->_`), así como nombres de sinónimos de tipos (ej. `String`, que es un renombre de `List Char`) y variables de tipos que representan parámetros (ej. las variables `a` y `b` en el tipo `(a -> b) -> List a -> List b`). Además, un tipo puede estar dado como la aplicación de un tipo τ_1 a otro tipo τ_2. Por ejemplo el tipo `Int -> Bool` se representa internamente como `((_->_ Int) Bool)`. Un tipo puede ser una metavariable de tipos (`?X`, `?Y`, `?Z`, ...) que representa una *incógnita* que se puede instanciar durante el proceso de inferencia. El usuario no puede escribir un tipo que incluya metavariables de tipos. Las metavariables se introducen por el inferidor de tipos como parte de su mecanismo interno.

Un *tipo restringido* es un tipo acompañado de una secuencia (posiblemente vacía) de restricciones de clase para algunas variables de tipo. Por ejemplo, `a -> String {Show a}` podría ser el tipo de la función `show`.

Por último, un *esquema* de tipos es un tipo restringido en el que algunas variables de tipo se encuentran cuantificadas universalmente, es decir, con cuantificadores "$\forall$". Por ejemplo, $\forall$`a`. (`a` $\to$ `a`) es el esquema de tipos correspondiente a la función identidad. Notar que los cuantificadores sólo pueden aparecer afuera de una expresión de tipos. Por ejemplo, la expresión `Int` $\to (\forall\alpha.\alpha)$ no se puede escribir en la gramática de tipos definida más arriba.

En general, el algoritmo de inferencia de tipos Hindley–Milner infiere los tipos utilizando las siguientes tres estrategias:

1. **Unificación.** Dados dos tipos concretos τ_1 y τ_2, el algoritmo de *unificación* instancia las metavariables presentes en τ_1 y τ_2 para *igualar* en estas dos expresiones de tipos. El algoritmo de unificación trabaja expandiendo primero las definiciones de todos los sinónimos de tipos. Por ejemplo, si se quiere unificar el tipo `List ?X` con el tipo `String`, se expande primero `String` a `List Char`, y a continuación se instancia `?X` en `Char`.

2. **Generalización.** Dado un tipo concreto τ, el procedimiento de *generalización* lo convierte en un esquema de tipos, cuantificando universalmente todas las variables libres que aparezcan en τ. Por ejemplo, el tipo concreto

`a` $\to$ `List (a, b)` se generaliza al esquema $\forall$`a`. $\forall$`b`. (`a` $\to$ `List (a, b)`). Una variante del procedimiento de generalización generaliza las *metavariables* de un tipo. Por ejemplo `?X` $\to$ `?X` se puede generalizar al esquema $\forall$`a`. (`a` $\to$ `a`).

Además, un requisito técnico es que las variables o metavariables que se generalizan no pueden aparecer en contextos de tipado fuera del *scope* actual. Por ejemplo, en esta definición de la identidad, `x` tiene tipo `?X`, y dentro del scope anidado la variable `z` también tiene tipo `?X`:

```
id x = let z = x in z
```

sin embargo no se puede generalizar el tipo de `z`, es decir no se puede generalizar la variable `?X` para asociarle el esquema ($\forall$`a`. `a`) a la variable `z`.

3. **Instanciación.** El procedimiento de *instanciación* recibe un esquema de tipos e instancia todas las variables que se encuentren universalmente cuantificadas, convirtiéndolas en metavariables frescas. Por ejemplo, el esquema $\forall$`a`. $\forall$`b`. (`a` $\to$ `b` $\to$ `a`) se instancia en el tipo concreto (`?X` $\to$ `?Y` $\to$ `?X`), donde `?X` e `?Y` son metavariables frescas.

Usando estas tres estrategias, el mecanismo general de inferencia de tipos se basa en las siguientes ideas:

1. Se dispone de un entorno que le asocia un esquema de tipos a cada identificador, incluyendo constructores y variables locales. Por ejemplo, a la lista vacía `[]` se le asocia el esquema de tipos $\forall$`a`. `List a`.

2. Cada vez que se usa un constructor o variable, se busca el esquema de tipos que tiene asociado en el entorno y se **instancian** sus metavariables. Por ejemplo, cada vez que se encuentra la lista vacía `[]` en una subexpresión, se instancia su esquema, creando una metavariable fresca `?X` y dándole tipo `List ?X`.

3. Cada vez que se encuentra una aplicación, o cualquier otra expresión que imponga restricciones sobre los tipos de las subexpresiones, se **unifican** los tipos que corresponda unificar. Por ejemplo, si se encuentra la aplicación `(e1 e2)`, donde el tipo de `e1` es τ_1 y el tipo de `e2` es τ_2, se unifica τ_1 con (τ_2 $\to$ `?X`) donde `?X` es una metavariable fresca que representa el tipo de la aplicación.

4. Cada vez que se encuentra una declaración local, ya sea como una ecuación en el programa principal, o como una expresión `(let x = e1 in e2)` o como una cláusula de la forma `(e2 where x = e1)`, se infiere el tipo de

la subexpresión `e1` y se **generaliza** su tipo concreto a un esquema E. A continuación, se infiere el tipo de la subexpresión `e2` en el entorno en el que la variable `x` está asociada al esquema E.

El mecanismo de inferencia de Ñuflo parte de estos fundamentos, pero es bastante más complejo, por dos razones. La primera razón es que se debe contemplar la presencia de definiciones **recursivas**. La segunda razón es que el inferidor de tipos debe **resolver las restricciones** de *typeclasses*, seleccionando las *instancias* que implementan la clase en cuestión para los tipos concretos que aparezcan cada vez que se usa un método de la clase.

Una característica clave del algoritmo de inferencia de tipos de Ñuflo es que **no solamente infiere los tipos de las subexpresiones** resolviendo las restricciones de *typeclasses*, sino que además **convierte el programa con typeclasses a un programa equivalente que no hace uso de typeclasses**. Para esto se siguen las ideas de Wadler y Blott [**?**], utilizando de la técnica conocida como *pasaje de diccionarios*. La ilustraremos brevemente con un ejemplo:

```
class EqShow a where
  ==   : a → a → Bool
  show : a → String

instance EqShow Bool where
  == True True = True
  show True = "True"

instance EqShow (List a) {EqShow a} where
  == (Cons x xs) (Cons y ys) = (== x y) && == xs ys
  show (Cons x xs) = show x ++ show xs

test : String
test = show (== (True : []) (False : []))
```

Figura 1: Programa original con *typeclasses*

- En la Figura **??**, se define la clase `EqShow` que tiene un método `==` para comparar dos valores por igualdad y un método `show` para convertir un valor a `String`. Además, se instancia la clase `EqShow` para el tipo de datos de los booleanos, y para el tipo de datos de las listas de `a` (siempre y cuando `a` sea a su vez instancia de `EqShow`). Por último, se usa el método `==` para comparar dos listas de booleanos y el método `==` para mostrar el booleano resultante.

```
data class{EqShow} a where
  mk{EqShow} : (a → a → Bool) → (a → String) → class{EqShow} a

== :  class{EqShow} a → a → a → Bool
== (mk{EqShow} m1 m2) = m1

show :  class{EqShow} a → a → String
show (mk{EqShow} m1 m2) = m2

instance{EqShow}{Bool} : class{EqShow} Bool
instance{EqShow}{Bool} =
  let m1 True True = True in
  let m2 True = "True" in
    mk{EqShow} m1 m2

instance{EqShow}{List} : class{EqShow} a → class{EqShow} (List a)
instance{EqShow}{List} .instance{EqShow}{a} =
  let m1 (Cons x xs) (Cons y ys) =
     (== .instance{EqShow}{a}) x y &&
     (== (instance{EqShow}{List} .instance{EqShow}{a})) xs ys in
  let m2 (Cons x xs) =
     (show .instance{EqShow}{a}) x ++
     (show (instance{EqShow}{List} .instance{EqShow}{a})) xs in
    mk{EqShow} m1 m2

test : String
test = let b = instance{EqShow}{Bool} in
       let l = instance{EqShow}{List} instance{EqShow}{Bool} in
         show b ((== l) (True : []) (False : []))
```

Figura 2: Programa transformado sin *typeclasses*

- En la Figura **??** vemos la salida del inferidor de tipos, que convirtió el programa original la Figura **??** en un programa equivalente que no utiliza *typeclasses*. Vemos cómo la declaración de la clase `(EqShow a)` se convirtió en una declaración de un tipo de datos inductivo `(class{EqShow} a)` cuyo constructor recibe como parámetros las implementaciones de los métodos `==` y `show` (en algún orden fijo).

  Las declaraciones de métodos `==` y `show` se convirtieron en funciones que proyectan los dos campos de un registro de tipo `(class{EqShow} a)`.

  La declaración de la instancia `(EqShow Bool)` se convirtió en una declaración de un valor `instance{EqShow}{Bool}` de tipo `(class{EqShow} Bool)`, que construye un registro con las implementaciones de los métodos correspondientes para el tipo `Bool`.

  Análogamente, la declaración de la instancia `(EqShow (List a))` se convirtió en una declaración de un valor `instance{EqShow}{List}`. Notar que, como la instancia de `(EqShow (List a))` depende a su vez que que el parámetro de tipos `a` sea instancia de `EqShow`, `instance{EqShow}{List}` es una *función* de tipo `class{EqShow} a -> class{EqShow} (List a)`, cuyo parámetro corresponde a la instancia de `EqShow` para el tipo `a`.

  Por último, cuando se utilizan los métodos `==` y `show` en casos concretos, el inferidor de tipos *resuelve* las restricciones de clase. Así por ejemplo, cuando se comparan dos listas de booleanos por igualdad, se usa la función `==` pasándole como primer parámetro un valor concreto de tipo `EqShow (List Bool)`, construido como:

  ```
  (instance{EqShow}{List} instance{EqShow}{Bool})
  ```

4.3. Detalles de implementación

4.3.1. Representación de tipos

El sistema de tipos representa internamente los tipos a secas, tipos restringidos y esquemas de tipos con los siguientes tipos de datos de Haskell:

```
-- Tipos a secas:
data Type = TVar QName                    -- variable de tipos
          | TApp Type Type                -- aplicación
          | TMetavar TypeMetavariable     -- metavariable de tipos

-- Restricciones de clase:
data TypeConstraint = TypeConstraint QName Type

-- Tipos restringidos (acompañados de restricciones de clase):
data ConstrainedType = ConstrainedType [TypeConstraint] Type

-- Esquemas de tipos:
data TypeScheme = TypeScheme [QName] ConstrainedType
```

Los identificadores de las metavariables son simplemente enteros, es decir, `TypeMetavariable` es un renombre de `Int`. Por ejemplo, el tipo $a \to$ *?42* $\to$ *List (Tuple a Bool)* se representa con el siguiente valor de tipo `Type`:

```
TApp
  (TApp (TVar (Name "_→_")) (TVar (Name "a")))
  (TApp
    (TApp (TVar (Name "_→_")) (TMetavar 42))
    (TApp
      (TVar (Name "List"))
      (TApp
        (TApp (TVar (Name "Tuple")) (TVar (Name "a")))
        (TVar (Name "Bool"))
      )
    )
  )
```

Observar que la variable de tipo "_→_" representa el constructor de tipos función (o "tipo flecha") y está aplicada a dos argumentos, que representan el dominio y el codominio respectivamente. Además, notar que `a`, `List`, `Tuple` y `Bool` también son variables de tipo. En cambio, `?42` es una metavariable que se usa internamente para representar un tipo que aún no fue instanciado y cuyo identificador es el número `42`.

Como otro ejemplo, el esquema de tipos `a -> List b  { Eq a ; Show b }`, en el cual las variables `a` y `b` están cuantificadas universalmente, se representa con el siguiente valor de tipo `TypeScheme`:

```
TypeScheme [Name "a", Name "b"]
  (ConstrainedType
    [
      TypeConstraint (Name "Eq") (Name "a"),
      TypeConstraint (Name "Show") (Name "b")
    ]
    (TApp
      (TApp (TVar (Name "_→_")) (TVar (Name "a")))
      (TApp (TVar (Name "List")) (TVar (Name "b")))
    )
  )
```

4.3.2. Estado del inferidor de tipos

El inferidor de tipos se implementa a través de una mónada "`FailState`" que propaga un estado y permite además fallar emitiendo un mensaje de error. El estado interno del inferidor de tipos es un registro conformado por los siguientes campos:

- *Posición*: se mantiene permanentemente registro de la "posición actual" dentro del código fuente, destinada a reportar errores. La posición se obtiene del nodo del AST que se está procesando. Es decir, se cuenta con un campo:

  ```
  position :: Position
  ```

  Un valor de tipo `Position` tiene información sobre el nombre del archivo y el número de fila y columna en la que se encuentra la expresión que se está analizando.

- *Próximo identificador disponible*: el inferidor de tipos dispone de un contador que se incrementa cada vez que se necesita generar un nombre interno. Es decir, se cuenta con un campo que representa el próximo entero disponible para generar un nombre temporal:

  ```
  nextFresh :: Integer
  ```

  Por ejemplo, cuando se crea una variable de tipos fresca se le da un nombre como `"t{42}"`. Los nombres generados internamente incluyen llaves (`"{"` y `"}"`) para asegurar que el nombre no pueda ser ingresado por el usuario.

- *Tipos constantes*: el inferidor de tipos cuenta con un conjunto de nombres de tipos *constantes*:

  ```
  typeConstants :: Set QName
  ```

  Por ejemplo si se declara el tipo de datos `data Tree` el nombre `"Tree"` se incluye en este conjunto.

- *Sinónimos de tipos*: en una pasada preliminar, el inferidor de tipos recolecta las declaraciones de sinónimos de tipos en una tabla de sinónimos:

  ```
  typeSynonyms :: TypeSynonymTable
  ```

  la tabla es un diccionario que a cada sinónimo de tipos `T` que haya sido declarado en una declaración de la forma `type T x1 ... xn = expr` le asocia una tupla `([x1, ..., xn], expr)`.

- *Entorno*: el entorno es una estructura que, a cada variable, le asocia un esquema de tipos. Como en general los *scopes* pueden estar anidados, el entorno consta de una pila de "costillas"[1]. Cada *costilla* es un diccionario que asocia las variables en el *scope* local a sus respectivos esquemas de tipo:

  ```
  environment :: [Map QName TypeScheme]
  ```

- *Sustitución*: la sustitución es una estructura que almacena información sobre cómo se instancian las metavariables de tipo. Más precisamente, es un diccionario que, a cada identificador de metavariable de tipo, le asocia un tipo (a secas):

  ```
  substitution :: Map TypeMetavariable Type
  ```

  A medida que el inferidor resuelve problemas de unificación, determina el valor de algunas metavariables y refleja ese conocimiento extendiendo la sustitución.

- *Información sobre las clases y métodos*: en una pasada preliminar, el inferidor de tipos recolecta información sobre las declaraciones de clases, incluyendo el nombre del parámetro de cada clase:

  ```
  classParameters :: Map QName QName
  ```

  Para cada clase, se dispone de un diccionario que indica las signaturas de cada uno de sus métodos:

  ```
  classMethodSignatures :: Map QName (Map QName Signature)
  ```

  Además, por conveniencia, a cada método, se le asocia su *información*, que incluye el nombre de su clase y su signatura:

  ```
  methodInfo :: Map QName MethodInfo
  ```

[1] Tomamos la nomenclatura del inglés *rib*.

- *Instancias globales*: el estado del inferidor de tipos cuenta con un diccionario de *instancias globales*, es decir, instancias de clase definidas en el programa por el usuario. Más precisamente, es un diccionario:

  ```
  globalInstances :: Map (QName, QName) GlobalInstance
  ```

  que le asocia un valor a cada par de la forma $(C, cons)$ donde C es el nombre de una clase y *cons* es el nombre de un constructor de tipos que tiene declarada una instancia de la clase C. A este par se le asocia la correspondiente información sobre los métodos de clase definidos por la instancia de C para el constructor *cons*. Un valor de tipo `GlobalInstance` representa una instancia de una clase particular para algún constructor de tipos particular.

- *Próximo* placeholder *disponible*: el inferidor de tipos dispone de un contador que se incrementa cada vez que se necesita generar un nuevo *placeholder*:

  ```
  freshPlaceholder :: Integer
  ```

  Recordemos que los *placeholders* sirven para designar "huecos" en el código generado. Estos huecos representan instancias para restricciones clases todavía no resueltas y se completan en diferido a la hora de resolver las restricciones de clase.

- *Entorno de restricciones de clase*: el *entorno de restricciones de clase* es una estructura que sirve para propagar las restricciones de clase que se tienen sobre las metavariables (aún no instanciadas). Más precisamente, es un diccionario:

  ```
  constraintEnv :: Map (QName, TypeMetavariable) PlaceholderId
  ```

  que define un valor para cada par $(C, meta)$ donde C es el nombre de una clase y *meta* es el identificador de una metavariable que aún no está instanciada y a la cual se le impone esa restricción de clase. El valor correspondiente es un identificador de un *placeholder*.

  Cada vez que se usa una función polimórfica con restricciones de clase, las variables cuantificadas universalmente en su esquema de tipos se instancian en metavariables frescas. Observar que cada una de esas variables puede venir acompañada de restricciones de clase. Para cada metavariable fresca *meta* a la que corresponda una restricción de clase C, se extiende el entorno de restricciones de clase, asociando el par $(C, meta)$ a un nuevo placeholder.

  Por otra parte, cada vez que una *metavariable meta* se instancia en un tipo concreto, se aplica un mecanismo para resolver todas las restricciones de

la forma $(C, meta)$ que se encuentren en el entorno de restricciones. Dicho mecanismo falla si las restricciones no pueden ser satisfechas utilizando las declaraciones de instancia disponibles. Caso contrario, si el mecanismo tiene éxito, el *placeholder* se instancia quedando ligado a un término que representa la instancia correspondiente.

Observar que este mecanismo de resolución de restricciones a su vez puede llegar a crear nuevas restricciones y *placeholders*. Por ejemplo, supongamos que se cuenta con la declaración de instancia:

```
Eq (List a)   { Eq a }
```

y supongamos además que se tiene la restricción `(Eq, ?182)` que está asociada al placeholder `#56`. Si, bajo estas condiciones, la metavariable `?182` se instancia en `(List ?183)`, se crea un nuevo placeholder, digamos `#57`, que queda asociado a la restricción `(Eq, ?183)`.

- *Memoria de* placeholders: Los *placeholders* instanciados se asocian a expresiones en un diccionario:

```
placeholderHeap :: Map PlaceholderId Expr
```

5. Expansión — conversión al cálculo-$\lambda^{\mathtt{U}}$

En esta sección, se describe un mecanismo, llamado *expansión*, que convierte programas del lenguaje Ñuflo en términos del cálculo-$\lambda^{\mathtt{U}}$ equivalentes. Es decir, el término del cálculo-$\lambda^{\mathtt{U}}$ que se obtiene implementa el comportamiento esperado del programa fuente Ñuflo. En la siguiente sección, se describirá cómo se interpretan los términos del cálculo-$\lambda^{\mathtt{U}}$. La composición de estos dos componentes nos da un intérprete para Ñuflo.

La etapa de expansión requiere convertir un programa Ñuflo, representado como un nodo del AST, en una expresión del cálculo-$\lambda^{\mathtt{U}}$. Siguiendo la definición formal de la sintaxis del cálculo-$\lambda^{\mathtt{U}}$, los programas se definen inductivamente:

```
data Program = Fail
             | Alt Term Program
```

es decir, un programa es la alternativa de una lista de términos. Un término del cálculo-$\lambda^{\mathtt{U}}$ se representa con el tipo `Term`. Notar que los tipos `Term` y `Program` son mutuamente recursivos:

```
data Term = Var QName                              -- variable
          | Cons QName                             -- constructor
          | ConstInt Integer                       -- constante entera
          | ConstChar Char                         -- constante de caracter
          | Fresh QName Term                       -- introducción de variable fresca
          | Lam QName Program                      -- abstracción
          | LamL Location QName Program            -- abstracción alojada
          | Fix QName Term                         -- operador de punto fijo
          | App Term Term                          -- aplicación
          | Seq Term Term                          -- secuencia
          | Unif Term Term                         -- unificación
          | Function PrimitiveFunction [Term]      -- función primitiva
          | Command PrimitiveCommand [Term]        -- comando primitivo
```

Estrictamente hablando, el lenguaje objeto no es el cálculo-$\lambda^{\mathtt{U}}$ puro, sino el cálculo-$\lambda^{\mathtt{U}}$ extendido con: constantes enteras y de caracter, un operador de punto fijo explícito, además de funciones y comandos primitivos.

El proceso de conversión de un programa Ñuflo a uno del cálculo-$\lambda^{\mathtt{U}}$ está implementado como una función en Haskell:

$$\texttt{desugar : ÑufloProgram} \rightarrow \texttt{Term}$$

El proceso de expansión es un recorrido recursivo relativamente sencillo sobre el programa fuente. A continuación se describen los aspectos clave de este proceso:

1. **Combinar múltiples ecuaciones en una única ecuación.**
   En una pasada preliminar, antes de realizar la expansión propiamente dicha, se transforman todas las declaraciones de funciones que incluyen más de una ecuación en una nueva declaración que consta de una única ecuación que contiene a todas las anteriores.

   Supongamos que se tiene una función `f` con múltiples ecuaciones:

   $$\begin{aligned} \texttt{f args}_1 &= \texttt{body}_1 \\ \texttt{f args}_2 &= \texttt{body}_2 \\ &\vdots \\ \texttt{f args}_n &= \texttt{body}_n \end{aligned}$$

   Donde $\texttt{args}_i$ representa la lista de parámetros y $\texttt{body}_i$ representa el cuerpo de la ecuación numero i respectivamente.

   Como primer paso, se calcula el número máximo de argumentos que aparecen en la definición de `f`:

   $$\texttt{M} = \texttt{max(length(args}_n\texttt{), ..., length(args}_n\texttt{))}$$

Se produce la siguiente única ecuación para `f` utilizando la cantidad máxima de argumentos:

$$\begin{array}{rcl} \mathtt{f} = \lambda x_1 \ldots x_{\mathtt{M}} & \rightarrow & ((\lambda\, \mathtt{args}_1 \rightarrow \mathtt{body}_1)\, x_1 \ldots x_{\mathtt{M}}) \\ & \mid & ((\lambda\, \mathtt{args}_2 \rightarrow \mathtt{body}_2)\, x_1 \ldots x_{\mathtt{M}}) \\ & \vdots & \\ & \mid & ((\lambda\, \mathtt{args}_n \rightarrow \mathtt{body}_n)\, x_1 \ldots x_{\mathtt{M}}) \end{array}$$

Observar que el cuerpo de la abstracción más externa es una alternativa de n casos. La i-ésima ecuación de la declaración original corresponde a la i-ésima rama. De esta manera, todas la funciones de un programa Ñuflo se pueden manipular de manera uniforme, asumiendo que están declaradas a través de una sola ecuación.

2. **Expansión de una abstracciones lambda cuyos parámetros son patrones.**
   Ñuflo permite escribir λ-abstracciones cuyo parámetro es un patrón:

   $$\lambda\, \mathtt{pattern} \rightarrow \mathtt{body}$$

   Donde `pattern` es un patron y `body` es la expresión correspondiente al cuerpo de la abstracción. Sin embargo, en el cálculo-$\lambda^{\mathtt{U}}$ las abstracciones reciben como parámetro una variable. Una abstracción como la de arriba se expande de la siguiente manera:

   $$\lambda\, x \rightarrow \mathtt{fresh}\ y_1 \ldots y_n\ \mathtt{in}\ ((x \sim \mathtt{pattern})\ \mathtt{\&}\ \mathtt{body})$$

   Las variables $y_1, \ldots, y_n$ son las variables libres que aparecen en el patrón, y que se encuentran ligadas en el cuerpo. La abstracción instancia $y_1, \ldots, y_n$ en frescas, unifica el argumento x con el patrón *pattern*, lo cual, posiblemente, instancia las variables $y_1, \ldots, y_n$. En caso de que la unificación ($\_\sim\_$) tenga éxito, se devuelve el cuerpo original de la función (`body`).

   Notar que las abstracciones con varios parámetros no deben considerarse aparte, porque el proceso de análisis sintáctico ya las interpreta como múltiples abstracciones anidadas. Por ejemplo

   $$\lambda\, \mathtt{pattern}_1 \ldots \mathtt{pattern}_n \rightarrow \mathtt{body}$$

   se analiza sintácticamente igual que

   $$\lambda\, \mathtt{pattern}_1 \rightarrow \ldots\ \lambda\, \mathtt{pattern}_n \rightarrow \mathtt{body}$$

3. **Expansión de `let` no recursivo que define una variable.**
   Una expresión `let` de la forma:

   ```
   let x = expr in body
   ```

donde la variable `x` no ocurre en `expr`, se puede expandir a la siguiente expresión semánticamente equivalente, expresada en términos de la aplicación y la abstracción:

$$(\lambda \texttt{x} \rightarrow \texttt{body})\ \texttt{expr}$$

4. **Expansión de `let` que define una función.**
   Una expresión `let` de la forma:

   $$\texttt{let f } \texttt{p}_1 \texttt{ ... } \texttt{p}_n \texttt{ = expr in body}$$

   y donde $\texttt{p}_1, \ldots, \texttt{p}_n$ representan patrones contra los que se comparan los parámetros formales de la función `f`, se puede expandir a la siguiente definición:

   $$\texttt{let f = } (\lambda\ \texttt{p}_1 \texttt{ ... } \texttt{p}_n \rightarrow \texttt{expr) in body}$$

   Notar que si la variable `f` no aparece libre en `expr`, obtenemos un `let` no recursivo que define una variable. Así este caso se reduce inmediatamente al caso anterior y a la expansión de una abstracción lambda cuyos parámetros son patrones.

5. **Expansión del let recursivo.**
   En el caso más general de todos, una expresión `let` tiene la siguiente forma:

   $$\begin{array}{lll} \texttt{let} & \texttt{f}_1\ \vec{\texttt{p}}_1 & = \ \texttt{e}_1 \\ & \vdots & \\ & \texttt{f}_n\ \vec{\texttt{p}}_n & = \ \texttt{e}_n \\ \texttt{in} & \texttt{body} & \end{array}$$

   donde cada nombre de función $\texttt{f}_i$ puede tener varias ecuaciones asociadas que la definen. En esta declaración, $\vec{\texttt{p}}_i$ es una lista de patrones contra los cuales se comparan los parámetros formales de la función $\texttt{f}_i$. Además, notar que, en general, la variable $\texttt{f}_i$ puede ocurrir libre en la expresión $\texttt{e}_j$ para todo par $1 \leq i, j \leq n$.

   El primer paso de expansión es aplicar la transformación descripta más arriba para combinar múltiples ecuaciones en una única ecuación. Como resultado, podemos asumir que no hay nombres de funciones repetidas, es decir, $\texttt{f}_i \neq \texttt{f}_j$ si $i \neq j$.

   El segundo paso es aplicar para cada $1 \leq i \leq n$ la expansión de `let` que define una función, para convertirlo en un `let` que define una variable.

Como resultado, podemos asumir que la expresión tiene la siguiente forma:

$$\begin{array}{rll} \texttt{let} & \texttt{f}_1 & = \ \texttt{e}_1 \\ & \vdots & \\ & \texttt{f}_n & = \ \texttt{e}_n \\ \texttt{in} & \texttt{body} & \end{array}$$

La forma de convertir esta expresión a un término del cálculo-$\lambda^{\mathtt{U}}$ es "aplanando" la recursión mutua, construyendo una expresión `let` que define una única variable que está ligada a una tupla de n componentes cuya i-ésima componente corresponde al valor de la expresión $\texttt{e}_i$. Eso se define como:

$$\begin{array}{ll} \texttt{let} & \texttt{t} = (\texttt{fix t} \to \langle \texttt{e}_1^*, \ldots, \texttt{e}_n^* \rangle) \\ \texttt{in} & \texttt{body}^* \end{array}$$

donde `fix` es un operador de punto fijo[2] y $\texttt{x}^*$ es la expresión que se obtiene a partir de `x` ligando cada $\texttt{f}_i$ a la i-ésima componente de la tupla:

$$\begin{array}{rlll} \texttt{x}^* = & \texttt{let} & \texttt{f}_1 & = \ \pi_1(\texttt{t}) \\ & & \vdots & \\ & & \texttt{f}_n & = \ \pi_n(\texttt{t}) \\ & \texttt{in} & \texttt{x} & \end{array}$$

Notar que la operación para crear una tupla de n componentes $\langle x_1, \ldots, x_n \rangle$ y la proyección de su i-ésima componente $\pi_i^n(x)$ se pueden definir en el cálculo-$\lambda^{\mathtt{U}}$ de la siguiente manera, donde **tuple** es un constructor:

$$\begin{array}{rcl} \langle x_1, \ldots, x_n \rangle & = & \mathbf{tuple}\, x_1 \ \ldots \ x_n \\ \pi_i^n(x) & = & \texttt{fresh}\ x_1 \ \ldots \ x_n \ \texttt{in}\, (x \sim \mathbf{tuple}\, x_1 \ \ldots \ x_n)\, \texttt{\&}\, x_i \end{array}$$

6. **Expansión de la alternativa por casos.**
   Supongamos que se tiene una alternativa por casos de la forma:

$$\begin{array}{rll} \texttt{case e of} & \texttt{p}_1 \to \texttt{b}_1 \\ & \vdots & \\ & \texttt{p}_n \to \texttt{b}_n \end{array}$$

   donde `e` es la expresión a evaluar, $\texttt{p}_i$ es el patrón de la i-ésima rama de la alternativa y $\texttt{b}_i$ el cuerpo de dicha rama.

   La expresión de arriba se puede expandir a la aplicación:

$$\texttt{f e}$$

[2] El operador de punto fijo se podría definir igual que en el cálculo-λ, por ejemplo como $\texttt{fix} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$. Por cuestiones de eficiencia, en nuestra implementación del cálculo-$\lambda^{\mathtt{U}}$, se dispone de un operador de punto fijo primitivo.

donde `f` es la función definida como sigue:

$$\begin{array}{lcll} \mathtt{f} & = & \lambda \mathtt{x} & \to \ \mathtt{fresh}\ \vec{\mathtt{y}}_1\ \mathtt{in}\ ((\mathtt{x} \sim \mathtt{p}_1)\ \&\ \mathtt{b}_1) \\ & & & \mid \ \mathtt{fresh}\ \vec{\mathtt{y}}_2\ \mathtt{in}\ ((\mathtt{x} \sim \mathtt{p}_2)\ \&\ \mathtt{b}_2) \\ & & & \vdots \\ & & & \mid \ \mathtt{fresh}\ \vec{\mathtt{y}}_n\ \mathtt{in}\ ((\mathtt{x} \sim \mathtt{p}_n)\ \&\ \mathtt{b}_n) \end{array}$$

La lista $\vec{\mathtt{y}}_i$ incluye todas las variables libres del patrón $\mathtt{p}_i$. Esta función simula cada una de las ramas del `case` a través de n aplicaciones del operador de alternativa no determinística. Observar que las ramas no son mutuamente excluyentes.

6. Interpretación del cálculo-$\lambda^{\mathtt{U}}$

Para la evaluación de los programas en Ñuflo, en este trabajo se optó por un proceso de interpretación (definido de manera ingenua) en vez de por un proceso de compilación. Se tomó esta decisión porque contamos con una semántica operacional para el cálculo-$\lambda^{\mathtt{U}}$ que ya sabemos que es correcta (en el sentido de que es confluente). Implementar correctamente un compilador excede el alcance de este trabajo, porque para darle bases teóricas sólidas sería necesario definir una *máquina abstracta* para el cálculo-$\lambda^{\mathtt{U}}$.

El primer paso para evaluar un programa Ñuflo es expandir el programa a un término del cálculo-$\lambda^{\mathtt{U}}$ como se describió en la sección anterior.

A continuación, un término del cálculo-$\lambda^{\mathtt{U}}$ se evalúa de acuerdo con las reglas de semántica operacional del cálculo-$\lambda^{\mathtt{U}}$. En Ñuflo, al igual que en otros lenguajes lógicos con alternativa no determinística como Prolog, un programa puede tener muchos posibles resultados. En Prolog, la exploración del árbol de búsqueda sigue un recorrido *depth-first* (DFS). En la implementación de Ñuflo, en cambio, y al igual que en el caso de miniKanren, el árbol se recorre en orden *breadth-first* (BFS). Más precisamente, se sigue la estrategia de reducción Gross–Knuth [**?**, Def. 4.9.5], que consiste en tomar un programa y reducir al mismo tiempo todas las expresiones reducibles (*redexes*) que se encuentren presentes en él. Esto corresponde a aplicar la noción de reducción simultánea ($\Rightarrow$) definida en Definición **??**, eligiendo la segunda variante de una regla siempre que sea posible. Por ejemplo, se elige la regla $\mathtt{App}_2$ en lugar de la regla $\mathtt{App}_1$ siempre que esto sea posible.

Luego de realizar un paso de reducción simultánea para un programa, este resulta en un conjunto de *threads*. En este conjunto, puede haber *threads* que estén en *forma normal* y *threads* que no. Es decir, el programa P se puede escribir como $P \equiv (\oplus_{i=1}^{n} t_i) \oplus (\oplus_{j=1}^{m} s_j)$ donde los términos t_i se encuentran en forma normal y los términos s_j no se encuentran en forma normal. Los *threads* en forma normal son valores de tipo `IO` que serán "mostrados en pantalla",

o más precisamente sometidos a un proceso de *interacción*. Los *threads* que no estén en forma normal se volverán a someter al proceso de evaluación, que proseguirá indefinidamente hasta que el programa conste únicamente de *threads* en forma normal. Notar que cuando se ejecuta un paso de reducción simultánea, la cantidad de *threads* puede disminuir, aumentar o permanecer invariable.

Por ejemplo, si se tiene el siguiente término, donde suponemos que `c`, `d` y `e` son constructores:

```
(λ x → ((x ~ c) & c) | fresh y in ((x ~ d y) & y))
    ((λ x → x) (d e))
```

El término se evalúa de la siguiente manera:

1. En primer lugar, se alojan las dos abstracciones, con dos pasos simultáneos de `alloc`:

   ```
   (λ¹ x → ((x ~ c) & c) | fresh y in ((x ~ d y) & y))
       ((λ² x → x) (d e))
   ```

2. Se ejecuta un paso de `beta` en el argumento. La aplicación de más afuera no es reducible porque su argumento no es un valor:

   ```
   (λ¹ x → ((x ~ c) & c) | fresh y in ((x ~ d y) & y)) (d e)
   ```

3. Ahora se ejecuta el paso `beta` de la aplicación más externa. Notar que esto aumenta el número de *threads*, de 1 a 2:

   ```
   ((d e ~ c) & c) | fresh y in ((d e ~ d y) & y)
   ```

4. En el *thread* de la izquierda hay una unificación que falla, por lo tanto se aplica un paso de la regla `fail`. En el *thread* de la derecha, se introduce la variable fresca `y` con un paso de la regla `fresh`. Notar que esto reduce el número de *threads*, de 2 a 1:

   ```
   (d e ~ d y) & y
   ```

5. La unificación `d e ~ d y` tiene éxito, con un paso de la regla `unif`:

   ```
   ok & e
   ```

6. Por último, el término de la izquierda de la secuencia (`ok`) ya es un valor, por lo tanto con un paso de la regla `seq` se obtiene:

   ```
   e
   ```

Interacción con el entorno: ejecución de las formas normales. Como se mencionó antes, cuando un *thread* llega a encontrarse en forma normal, se lo somete a un proceso de "ejecución" o interacción. El programa principal en Ñuflo debe ser la función `main : () → IO`. El punto de entrada del intérprete de Ñuflo es la evaluación de la expresión `main ()`. Las formas normales de tipo `IO` se pueden entender como comandos o acciones que describen una interacción con el entorno. Cuando se ejecuta una forma normal de tipo `IO`, el resultado es un efecto de entrada/salida. No hay garantía sobre el orden en el que estas formas normales se ejecutan. Se garantiza que el intérprete de Ñuflo ejecuta todos los términos en forma normal que se puedan alcanzar a partir de la expresión `main ()`, en algún orden[3].

Como primitiva del lenguaje Ñuflo, se define el tipo de datos que sirve para representar interacciones con el entorno (`IO`) de la siguiente manera:

```
data IO where
end     : IO
print   : a → IO → IO
put     : String → IO → IO
get     : (String → IO) → IO
getChar : (Char → IO) → IO
getLine : (String → IO) → IO
```

El tipo `IO` puede verse como un conjunto de operaciones que realizan algún tipo de interacción con el entorno. A continuación se explica brevemente el comportamiento de cada constructor:

- `end`: operación nula, es decir, la operación que no tiene efecto.
- `(print` x io`)`: operación que escribe en la salida estándar el valor de x, en un formato apropiado. A continuación prosigue con la ejecución del comando io.
- `(put` x io`)`: operación que escribe en la salida estándar el *string* x. A continuación prosigue con la ejecución del comando io.
- `(get` f`)`: operación que lee el contenido de la entrada estándar completo hasta que se alcanza el final del archivo (es decir, hasta que se alcanza la condición de `end-of-file`) en un *string* x. A continuación prosigue con la evaluación y ejecución del comando (fx).
- `(getChar` f`)`: operación que lee un caracter x de la entrada estándar. A continuación prosigue con la evaluación y ejecución del comando (fx).

[3]El resultado de confluencia asegura que el multiconjunto de formas normales es único sin importar el orden en el que se evalúe el programa, pero la unicidad se debe entender módulo la relación de equivalencia estructural, que permite conmutar *threads*.

- `(getLine` f`)`: operación que lee una línea de la entrada estándar (es decir, hasta que se alcanza el final de línea, cuya representación exacta puede depender del entorno) en un *string* x. A continuación prosigue con la evaluación y ejecución del comando (fx).

Por ejemplo, la siguiente expresión representa una interacción con el entorno en la que se solicita al usuario que ingrese un *string*, se lee una línea de la entrada y se imprime en pantalla el reverso del *string* leído:

```
put "Ingrese un string:"
    (getLine (λ s → put (reverse s) end))
```

7. Tipos y operaciones primitivas

En esta sección se describen los tipos y operaciones primitivas de Ñuflo. Todos los tipos y operaciones primitivas de Ñuflo están declarados en el módulo `PRIM` cuyos nombres siempre están disponibles para el usuario, sin necesidad de importarlos explícitamente.

7.1. Tipos de datos

Los tipos de datos primitivos de Ñuflo son:

1. `_→_`: constructor de tipos para las relaciones/funciones.
2. `Int`: tipo de los enteros.
   Sus constructores se escriben `0, 1, 2, ...`
3. `Char`: tipo de los caracteres.
   Sus constructores se escriben `'0', '1', ..., 'A', 'B', ...`
4. `List`: tipo de las listas. Su definición es equivalente a esta:

```
data List a where
   []  :  List a
   _:_ :  a → List a → List a
```

5. `String`: tipo de los *strings*, renombre de `List Char`.
6. `()`: tipo unitario, que tiene un único constructor también notado `()`.

7. `IO`: tipo para representar las interacciones con el entorno, definido como ya se detalló arriba:

```
data IO where
end     : IO
print   : a → IO → IO
put     : String → IO → IO
get     : (String → IO) → IO
getChar : (Char → IO) → IO
getLine : (String → IO) → IO
```

7.2. Identificadores reservados

Los siguientes identificadores tienen una interpretación especial:

1. El símbolo "`_`" (*underscore*) representa un término fresco. Esto se puede usar para expresar un patrón comodín igual que en Haskell o Prolog, pero también puede aparecer en el lado *derecho* de una definición. Su significado es equivalente al de `(fresh x in x)` y tiene tipo arbitrario, es decir `_ : a` para todo `a`.

2. El identificador `main` está reservado para la función principal, debe definirse una sola vez en todo el programa y debe tener tipo `main : () → IO`.

7.3. Operaciones primitivas

- **Falla.** Se nota con un término `fail : a`. La expresión `fail` es equivalente a la expresión `(case _ of {})`.
- **Alternativa.** Se nota con el operador binario `_|_ : a → a → a`. La expresión `(x | y)` es equivalente a `(case _ of { _ → x ; _ → y })`.
- **Secuencia.** Se nota con el operador binario `_&_ : a → b → b`.
- **Unificación.** Se nota con el operador binario `_~_ : a → a → ()`.

7.4. Tabla de asociatividad y precedencia

Los siguientes son los operadores primitivos que reconoce el analizador sintáctico de Ñuflo, con sus respectivas asociatividades y niveles de precedencia.

Operador	Descripción	Asociatividad	Nivel de precedencia
_→_	Tipo funcional	a derecha	50
_|_	Alternativa	a derecha	60
_;_	Secuencia	a derecha	70
_∼_	Unificación	a derecha	80
_:_	Constructor de listas	a derecha	90

8. Ejemplos de programas Ñuflo

El objetivo de esta sección es mostrar algunos ejemplos de programas en Ñuflo. Para comenzar, se modelan los números naturales y la operación de adición por inducción en la estructura de los números:

```
data Nat where
  Z  : Nat
  S  : Nat → Nat

_+_ : Nat → Nat → Nat
Z   + n = n
S n + m = S (n + m)
```

Los números naturales se representan en este ejemplo con la codificación de Peano. Por ejemplo, el 3 se representa con el valor `S (S (S Z))`. El operador infijo _+_ define una relación funcional que recibe dos números naturales y retorna la suma de ellos. Se define haciendo *pattern matching* sobre la estructura de los números a sumar. Desde el punto de vista práctico, este programa está incompleto, porque no cuenta con un punto de entrada (función `main`). Recordar que la función `main` debe ser de tipo `() → IO`. La función *main* podría ser, por ejemplo:

```
main () = print (
            fresh x in
              (S (S Z) + x) ~ S (S (S Z))
              & x
           ) end
```

Observar que ahora el comportamiento de este programa es el siguiente: se instancia una variable simbólica fresca `x`, se evalúan las expresiones `(S (S Z) + x)` y `(S (S (S Z)))`, y se unifican sus resultados. Si la unificación tiene éxito, se retorna el valor en el que haya sido instanciada la variable `x`. En este caso, la unificación tiene éxito y la variable `x` se instancia en el valor `S Z`, que representa el número 1. El comando `print` escribe dicho resultado en la salida estándar.

Siguiendo las mismas ideas, es posible definir la relación que representa la sustracción (_−_) de la siguiente manera:

```
_-_ : Nat → Nat → Nat
x - y = fresh z in
                (y + z) ~ x
                & z
```

Notar que, a diferencia de la suma, la resta no se define usando *pattern matching* sino a través del operador de unificación, devolviendo una variable `z` tal que `y + z` unifique con x. No es necesario definir esta función por casos ya que al utilizar la operación de adición, se cubren todos los posibles casos. Por lo tanto, evaluar:

```
main () = print (S (S (S Z)) - S Z) end
```

se obtiene como resultado `S (S Z)`, la representación del numero 2.

Por otro lado, haciendo uso de la alternativa no deterministica, podríamos definir la función `coin` que no determinísticamente devuelve 1 ó 2:

```
coin : () → Nat
coin () = S Z | S (S Z)

main () = print (coin () + coin ()) end
```

En este caso, el programa escribe en la salida *todas* las posibles sumas de los resultados de `coin`. Esto es:

`S (S Z)`	$(1+1)$
`S (S (S Z))`	$(1+2)$
`S (S (S Z))`	$(2+1)$
`S (S (S (S Z)))`	$(2+2)$

Notar que se incluyen resultados repetidos en la salida. Tener en cuenta también que no hay ninguna garantía sobre el orden en el que se escriben estos resultados en la salida.

Como ejemplo ilustrativo, veremos cómo implementar algunas operaciones sobre listas. Recordar que las listas son un tipo de dato primitivo en Ñuflo, por lo que no es necesario definirlo previamente (con constructores `[]` y `_ : _`). Se implementa la función *append* (`_++_`) recursivamente haciendo *pattern matching* sobre la estructura de las listas:

```
_++_ : List a → List a → List a
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Una vez definida la operación para concatenar dos listas, podemos definir la relación *prefijo*, que relaciona una lista con todos sus prefijos, de manera sumamente declarativa:

```
prefijo : List a → List a
prefijo (xs ++ ys) = xs

main () = print (prefijo (1 : 2 : 3 : [])) end
```

Al evaluar este programa se obtienen como resultado todos los posibles prefijos de la lista `(1 : 2 : 3 : [])`:

```
[]
1 : []
1 : 2 : []
1 : 2 : 3 : []
```

Observar que la relación `prefijo` está definida, aparentemente, haciendo *pattern matching* sobre un argumento que resulta de aplicar la función (_++_) definida previamente. A diferencia de Haskell, donde los patrones están limitados a estar formados por aplicaciones de constructores, en Ñuflo un patrón puede ser una expresión arbitraria. Recordar que la definición de `prefijo` de arriba es equivalente a la siguiente, como se describió en la sección que describe la etapa de expansión:

```
prefijo : List a → List a
prefijo lista = fresh xs ys in (lista ~ (xs ++ ys)) & xs
```

Utilizando esta misma técnica (con patrones funcionales), es posible definir la función `last`, que devuelve el último elemento de una lista no vacía, de manera declarativa, como sigue:

```
last : List a → a
last (_ ++ (x : [])) = x

main () = print (last (1 : 2 : 3 : [])) end
```

Como es de esperar, el resultado de este es el número 3, qué es el único valor para el cual la unificación tiene éxito.

Haciendo uso de la exploración exhaustiva implícita que surge al evaluar la alternativa no determinística dada por una relación definida por medio de varias ecuaciones, se puede definir la relación `inter`, que dado un elemento `x` y una lista `l`, genera todas las posibles listas que resultan de intercalar `x` en algún lugar de `l`:

```
inter : a → List a → List a
inter x []       = x : []
inter x (y : ys) = x : (y : ys)
inter x (y : ys) = y : inter x ys

main () = print (inter 42 (1 : 2 : 3 : [])) end
```

Este programa tiene cuatro resultados:

```
42 : 1 : 2 : 3 : []
1 : 42 : 2 : 3 : []
1 : 2 : 42 : 3 : []
1 : 2 : 3 : 42 : []
```

Usando la relación `inter` y la función `foldr` usual, es posible definir una nueva relación `perm` que genera todas las posibles permutaciones de una lista:

```
foldr : (a → b → b) → b → List a → b
foldr _ z []       = z
foldr f z (x : xs) = f x (foldr f z xs)

perm : List a → List a
perm xs = foldr inter [] xs

main () = print (perm (1 : 2 : 3 : [])) end
```

Esto da como resultado:

```
1 : 2 : 3 : []
1 : 3 : 2 : []
2 : 1 : 3 : []
2 : 3 : 1 : []
3 : 1 : 2 : []
3 : 2 : 1 : []
```

es decir, todas las posibles permutaciones de la lista `(1 : 2 : 3 : [])`.

El uso de herramientas como la búsqueda exhaustiva y la unificación permiten entender un programa como un conjunto de definiciones de datos y relaciones entre ellos. Como se mencionó antes, los patrones en Ñuflo pueden ser expresiones arbitrarias. Usando esta técnica, es posible definir una relación de orden superior `inv`, que dada una relación binaria devuelve la relación inversa. Desde el punto de vista denotacional, `inv` relaciona a cada relación binaria R con la relación $R^{-1} = \{(b, a) \mid (a, b) \in R\}$:

```
inv : (a → b) → b → a
inv f (f x) = x
```

Notar que la segunda ocurrencia de `f` en el lado izquierdo de la ecuación está ligada por la primera ocurrencia de `f`. De hecho, la definición de arriba se convierte en la siguiente definición durante la etapa de expansión:

```
inv : (a → b) → b → a
inv = λ f → λ y → fresh x in (y ~ (f x)) & x
```

Por ejemplo, consideremos el tipo de datos `Simpson` y la función `padre`:

```
data Simpson where
  Abe    : Simpson
  Homero : Simpson
  Bart   : Simpson
  Lisa   : Simpson
  Maggie : Simpson

padre : Simpson → Simpson
padre Homero = Abe
padre Bart   = Homero
padre Lisa   = Homero
padre Maggie = Homero
```

Notemos de paso que la función `padre` también se podría definir usando el operador de alternativa no deterministica en el patrón:

```
padre : Simpson → Simpson
padre Homero                 = Abe
padre (Bart | Lisa | Maggie) = Homero
```

A continuación, se podría definir la relación `hijo` como sigue:

```
hijo (padre x) = x
```

Este es precisamente el patrón que abstrae la relación de orden superior `inv`. En efecto, aplicando la relación `inv` sobre la relación `padre` se podría definir la relación `hijo` directamente como:

```
hijo = inv padre
```

Por ejemplo, el programa que tiene como resultado a todos los nietos de `Abe` se podría definir de la siguiente manera:

```
abuelo  = padre ∘ padre
nieto   = inv abuelo
main () = print (nieto Abe) end
```

donde `_∘_` representa la composición de relaciones, definida de la manera usual como `_∘_ f g x = f (g x)`. Esto da como resultado:

```
Bart
Lisa
Maggie
```

Como último ejemplo, se implementa un inferidor de tipos minimal para el cálculo-λ simplemente tipado. Los términos se representan con la codificación basada en índices de de Bruijn, es decir, las variables no se representan con nombres, sino con un número natural que indica el índice de la abstracción que la liga. Por ejemplo, el término $\lambda x.\,(\lambda y.\,(y\,x))\,x$ se codifica como $\lambda((\lambda\,0\,1)\,0)$. Para esto, se definen los siguientes tipos de datos:

```
data Id where
  Z : Id
  S : Id → Id

data Type where
  BOOL : Type
  _=>_ : Type → Type → Type

data Ctx where
   ∅  : Ctx
  _,_ : Ctx → Type → Ctx

data Term where
  var : Id → Term
  lam : Term → Term
  app : Term → Term → Term
```

El tipo de datos `Id` representa los índices de de Bruijn. El tipo de datos `Type` representa los tipos del cálculo-λ simplemente tipado con un único tipo básico (`BOOL`). El tipo de datos `Ctx` sirve para representar los contextos de tipado. Un contexto de tipado Γ puede ser vacío o, recursivamente, de la forma Γ, A donde A es un tipo. El tipo de datos `Term` representa los términos del cálculo-λ simplemente tipado.

A continuación se define una relación $\_\ni\_::\_$ de la siguiente manera:

```
_∋_::_ : Ctx → Id → Type → ()
(Γ , A) ∋ Z   :: A = ()
(Γ , A) ∋ S x :: B = Γ ∋ x :: B
```

Esta relación está definida de tal modo que valga la siguiente igualdad:

$$(\varnothing, A_{n-1}, A_{n-2}, \ldots, A_1, A_0) \ni i :: A_i = ()$$

Usando esta definición, se puede definir otra relación _⊢_::_ que corresponde a la relación de tipabilidad:

```
_⊢_::_ : Ctx → Term → Type → ()
Γ ⊢ var x   :: A = Γ ∋ x :: A
Γ ⊢ lam t   :: (A => B) = (Γ , A) ⊢ t :: B
Γ ⊢ app t s :: B = fresh A in
                            Γ ⊢ t :: (A => B)
                          & Γ ⊢ s :: A
```

Observar que a una variable `x` se le puede asignar el tipo `A` si y sólo si el índice `x` del contexto Γ es el tipo `A`. Una abstracción `lam t` tiene tipo `A => B` si y sólo si el término `t` tiene tipo `B` en el contexto Γ, `A`. Una aplicación `app t s` tiene tipo `B` en el contexto Γ si y sólo si el término `t` tiene tipo `A => B` en Γ y el término `s` tiene el tipo `A` en Γ, para algún tipo `A` fresco.

Por ejemplo, el siguiente programa, instancia una variable simbólica fresca `A` que representa un tipo. A continuación, exige que se verifique el juicio de tipado ∅ ⊢ `lam (var Z)` :: `A`, es decir, exige que `A` sea el tipo de la función identidad en el contexto vacío. Por último, devuelve el tipo `A`:

```
main () = print (
                fresh A in
                  (∅ ⊢ lam (var Z) :: A)
                  & A
                ) end
```

El resultado es el tipo de la identidad. En una ejecución concreta se obtiene por ejemplo `?{26964} => ?{26964}`, donde `?{26964}` es una variable fresca que representa un tipo desconocido. Entonces, esto proporciona un inferidor de tipos donde, dado un término y una variable de tipo como incógnita, se calcula el valor del tipo correspondiente a este término.

La misma definición podría ser utilizada a la inversa para buscar exhaustivamente programas que tengan un cierto tipo. Por ejemplo, el programa:

```
main () = print (
               fresh t in
                 (∅ ⊢ t :: (BOOL => BOOL))
                 & t
               ) end
```

Genera todos los posibles términos que tienen tipo `(BOOL => BOOL)` en el contexto vacío. Un posible resultado de este programa es el término `(lam (var Z))`. Por la manera en que está escrito el programa, después de producir este resultado, Ñuflo sigue buscando otros posibles programas del mismo tipo. Esta búsqueda nunca finaliza.