# Lógica Computacional, 2019-2
# Nota de laboratorio 7*

## Sintaxis de la Lógica de Primer Orden

Manuel Soto Romero

20 de marzo de 2019

Facultad de Ciencias, UNAM

Al igual que la Lógica Proposicional, y cualquier sistema lógico, la Lógica de Primer Orden cuenta con un lenguaje formal que se usa como medio de expresión mismo al que se conoce como sintaxis. En esta nota se estudia la sintaxis de la Lógica de Primer Orden desde un punto de vista complementamente práctico, a través de su definición en el lenguaje de programación Haskell.

## 1. El lenguaje de la Lógica de Primer Orden

A diferencia de la Lógica Proposicional, el lenguaje de la Lógica de Primer Orden, también llamada Lógica de Predicados, no tiene una única representación, ésta dependerá de la estructura semántica que se requiera para representar objetos y la representación entre estos. De esta forma, el alfabeto que conforma a la Lógica de Primer Orden consta de una parte común a todos los lenguajes y una parte particular llamada semejanza o signatura del lenguaje.

La parte común a todos los lenguajes consta de:

- Un conjunto infinito de variables $VAR = \{x_1, ..., x_n, ...\}$.
- Constantes lógicas: $\bot, \top$
- Conectivos lógicos: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- Cuantificadores: $\forall, \exists$.
- Símbolos auxiliares como paréntesis y comas.
- Si se agrega el símbolo de igualdad ($=$) se dice que el lenguaje tiene igualdad.
- La signatura de un lenguaje en particular está dada por:
  - ▷ Un conjunto $P$ de símbolos de predicado: $P_1, ..., P_n, ....$ A cada símbolo se le asigna un índice o número de argumentos $m$, el cual se hace explícito escribiendo $P_n^{(m)}$que indica que el símbolo de predicado $P_n$ necesita $m$ argumentos.
  - ▷ Un conjunto $F$, se símbolos de función: $f_1, ..., f_n, ...$ Análogamente a los símbolos de predicado, los símbolos de función tienen asignado un índice $m$ para indicar el número de argumentos que necesita la función.

*Basado en las *Notas de clase para el curso de Lógica Computacional* del Dr. Favio E. Miranda, et al., revisión 2017-2.

▷ Un conjunto $C$ de símbolos de constante: $c_1, ..., c_n, ...$

De esta forma, un Lenguaje de Primer Orden $L$ queda determinado de manera única por su signatura, es decir:

$$L = P \cup F \cup C$$

## 2. Términos

Los términos del lenguaje son aquellas expresiones que representan objetos en la semántica y su definición es:

**Definición 1.** Un término es alguno de los siguientes:

- Los símbolos de constante $c_1, ..., c_n, ...$ son términos.
- Las variables $x_1, ..., x_n, ...$ son términos.
- SI $f^{(m)}$ es un símbolo de función y $t_1, ..., t_m$ son términos, entonces $f(t_1, ..., t_m)$ es un término.
- Son todos.

Se llama `TERM` al conjunto de términos de un lenguaje. El Listado de código 1 muestra la definición de este conjunto (tipo de dato) en Haskell.

```haskell
-- Definición del conjunto TERM.
data TERM = Var String
          | Fun String [TERM]
          deriving(Eq)
```

Listado de código 1: Representación del conjunto TERM en Haskell

*Observación* 1. Se omite un constructor para representar constantes pues es posible su representación mediante funciones sin argumentos. Por ejemplo, la constante `c1` se representa mediante `Fun "c1" []`.

*Observación* 2. Se deriva al tipo de dato de la clase `Eq` para poder hacer comparaciones con sus valores más adelante.

Para imprimir términos en la terminal se hace a `TERM` parte de la clase de tipos `Show`. El Listado de código 2 muestra esta implementación.

```haskell
instance TERM Show where
   show (Var v)    = v
   show (Fun n ls) = n ++ "(" ++ aux ls ++ ")"

-- Función auxiliar.
-- Agrupa, en formato cadena, los argumentos de una función separados
-- por coma.
aux :: [TERM] -> String
aux []     = ""
aux [x]    = show x
aux (x:xs) = show x ++ ", " ++ aux xs
```

Listado de código 2: TERM como miembro de Show

# 3. Fórmulas

## 3.1. Fórmulas atómicas

El siguiente paso consiste en determinar las fórmulas atómicas, dadas por:

- Las constantes lógicas: $\bot, \top$
- Las expresiones de la forma: $P_1(t_1, ..., t_n)$ donde $P_1$ es un predicado y $t_1, ..., t_n$ son términos.
- Las expresiones de la forma: $t_1 = t_2$ si el lenguaje cuenta con igualdad.

Se llama `ATOM` al conjunto de fórmulas atómicas de un lenguaje. El Listado de código 3 muestra la definición de este conjunto (tipo de dato) en Haskell.

```haskell
-- Definición del conjunto ATOM.
data ATOM = Cte Bool
          | Prd String [TERM]
          | Eq TERM TERM
          deriving(Eq)
```

Listado de código 3: Representación del conjunto ATOM en Haskell

*Observación* 3. Se deriva al tipo de dato de la clase `Eq` para poder hacer comparaciones con sus valores más adelante.

Para imprimir términos en la terminal se hace a `ATOM` parte de la clase de tipos `Show`. El Listado de código 4 muestra esta implementación.

```haskell
instance ATOM Show where
   show (Cte b)    = show b
   show (Prd n ls) = n ++ "(" ++ aux ls ++ ")"
   show (Eq p q)   = show q ++ " = " ++ show q
```

Listado de código 4: ATOM como miembro de Show

*Observación* 4. Se usa en el Listado de código 4 la función auxiliar `aux` definida en el Listado de código 2.

## 3.2. Fórmulas en Lógica de Primer Orden

El conjunto `PO` también llamado `FORM` de expresiones compuestas aceptadas en un lenguaje $L$, llamadas fórmulas, se define recursivamente como sigue:

- Si $\varphi \in ATOM$, entonces $\varphi \in PO$.
- Si $\varphi, \psi \in PO$, entonces $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \to \psi), (\varphi \leftrightarrow \psi) \in PO$.
- Si $\varphi \in PO$ y $x \in VAR$, entonces $(\forall x \varphi), (\exists x \varphi) \in PO$.
- Son todas.

El Listado de código 5 muestra la implementación del conjunto de fórmulas de la Lógica de Primer Orden en Haskell:

```haskell
-- Definición del lenguaje PO de la Lógica de Primer Orden
data PO = FA ATOM
        | Neg PO
        | Conj PO PO
        | Disy PO PO
        | Impl PO PO
        | Syss PO PO
        | Pt String PO
        | Ex String PO
```

Listado de código 5: Representación del conjunto ATOM en Haskell

Para imprimir fórmulas en la terminal se hace a `PO` parte de la clase de tipos `Show`. El Listado de código 6 muestra esta implementación.

```haskell
instance PO Show where
   show (FA a)     = show a
   show (Neg p)    = "no " ++ show p
   show (Conj p q) = "(" ++ show p ++ " & " ++ show q ++ ")"
```

```
5     show (Disy p q) = "(" ++ show p ++ " | " ++ show q ++ ")"
6     show (Impl p q) = "(" ++ show p ++ " -> " ++ show q ++ ")"
7     show (Syss p q) = "(" ++ show p ++ " <-> " ++ show q ++ ")"
8     show (Pt x p)   = "Pt" ++ x ++ " " ++ show p
9     show (Ex x p)   = "Ex" ++ x ++ " " ++ show p
```

Listado de código 6: PO como miembro de Show

## 4. Definiciones recursivas

Para definir funciones recursivas sobre términos y fórmulas se usan los siguientes principios:

**Definición recursiva para Términos** Para definir una función $h : TERM \to A$, basta definir $h$ como sigue:

- Definir $h(x)$ para $x \in VAR$.
- Definir $h(c)$ para $c \in C$.
- Suponiendo que $h(t_1), ..., h(t_n)$ están definidas, definir $h(f(t_1, ..., t_n))$ para $f \in F$ de índice $n$.

**Definición recursiva para Fórmulas** Para definir una función $h : PO \to A$, basta definir $h$ como sigue:

- Definir $h$ para cada fórmula atómica.
- Suponiendo definidas $h(\varphi)$ y $h(\psi)$, definir a partir de ellas a $h(\neg\varphi), h(\varphi \wedge \psi), h(\varphi \to \psi), h(\varphi \leftrightarrow \psi), h(\forall x\varphi)$ y $h(\exists x\varphi)$.

**Ejemplo 1.** Definir la función `subterminos` tal que `(subterminos t)` representa la lista de subtérminos de un término `t`, sin repeticiones. Esto es:

- $subterminos(x) = \{x\}$
- $subterminos(c) = \{c\}$
- $subterminos(f(t_1, ..., t_n)) = \{f(t_1, ..., t_n)\} \cup subterminos(t_1) \cup ... \cup subterminos(t_n)$

**Solución.** Función `subterminos`:

```haskell
import Data.List

-- Función que obtiene los subtérminos de un término.
subterminos :: TERM -> [TERM]
subterminos (Var v)    = [(Var v)]
subterminos (Fun c []) = [(Fun c [])]
subterminos (Fun f ls) = [(Fun f ls)] ++ subtAux ls

-- Función auxiliar.
-- Obtiene los subtérminos de una lista.
subtAux :: [TERM] -> [TERM]
subtAux []     = []
subtAux (x:xs) = union (subterminos x) (subtAux xs)
```

Listado de código 7: Subtérminos de un término

*Observación* 5. La función subtAux (líneas 11 a 13) realiza la unión de subtérminos de la lista de términos de una función.

□

**Ejemplo 2.** Definir la función peso, tal que (peso p) es el peso de una fórmula, es decir, el número de conectivos y cuantificadores de una fórmula. Esto es:

- $peso\,(\bot) = peso\,(\top) = 0$
- $peso\,(P\,(t_1, .., t_n)) = 0$
- $peso\,(t_1 = t_2) = 0$
- $peso\,(\neg\varphi) = 1 + peso\,(\varphi)$
- $peso\,(\varphi \wedge \psi) = 1 + peso\,(\varphi) + peso\,(\psi)$
- $peso\,(\varphi \vee \psi) = 1 + peso\,(\varphi) + peso\,(\psi)$
- $peso\,(\varphi \to \psi) = 1 + peso\,(\varphi) + peso\,(\psi)$
- $peso\,(\varphi \leftrightarrow \psi) = 1 + peso\,(\varphi) + peso\,(\psi)$
- $peso\,(\forall x\varphi) = peso\,(\exists x\varphi) = 1 + peso\,(\varphi)$

**Solución.** Función peso:

```haskell
-- Función que calcula el peso de una fórmula.
peso :: PO -> Int
peso (FA a)      = 0
peso (Neg p)     = 1 + peso p
peso (Conj p q) = 1 + peso p + peso q
peso (Disy p q) = 1 + peso p + peso q
peso (Impl p q) = 1 + peso p + peso q
peso (Syss p q) = 1 + peso p + peso q
peso (Pt _ p)    = 1 + peso p
peso (Ex _ p)    = 1 + peso p
```

Listado de código 8: Peso de una fórmula

□