Dedicatoria

A mi esposa e hijo, consecuencia lógica en la ecuación del amor.

Paulo González G.

Talca, 2018

Agradecimientos

Este apunte es construido luego de varios años de trayectoria y como fruto de aquellos que influyeron tanto en el pregrado y postgrado.

Agradezco a Dios por mostrar el camino a seguir y poner en la ruta a quienes entregan las herramientas que orientan e iluminan en el sendero de la vida.

Índice

Índice de Figuras

1. Lógica

En esta parte del apunte, la sintáxis y semántica de la lógica de primer orden son presentados. Se comienza con un repaso de la lógica proposicional para proseguir con la lógica de primer orden. En ambos casos se analiza la consecuencia lógica y la herramienta de resolución para su establecimiento. Finalmente, se presenta una aplicación de la lógica para la prueba formal de programas cuando existe el establecimiento de las pre condiciones, invariantes y las postcondiciones.

1.1. Lógica Proposicional

La lógica de proposiciones es un formalismo lógico en el cual se representa cada oración o cada frase del lenguaje por una letra o símbolo donde un ente atómico representa algo que puede ser verdadero o falso.

1.1.1. Sintaxis de un lenguaje proposicional

El lenguaje esta formado por la unión de:

- Conectivos lógicos $\{\neg, \wedge, \vee, \supset, \equiv\}$.
- Simbolos de puntuacion $\{(,)\}$.
- Un conjunto P de letras proposicionales.

1.1.2. Definición de fórmula bien formada

La definición de fórmula viene dada por:

- Si $p \in P$, entonces p es una fbf (atómica).
- Si φ es fbf, entonces $\neg\varphi$ es fbf.
- Si φ y ψ son fbf, entonces $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \supset \psi)$ y $(\varphi \equiv \psi)$ son fbf.
- no hay más fbf.

Por lo tanto, tenemos una cantidad infinita de fórmulas aunque tengamos una cantidad finita de letras en el lenguaje.

1.1.3. Valor de verdad $\sigma(\varphi)$ de una fórmula φ:

- $\sigma(\varphi)$ es el mismo valor entregado por la asignación, ssi, φ es $p \in P$.
- 0 si φ es $\neg\psi$ y $\sigma(\psi) = 1$
- 1 si φ es $\neg\psi$ y $\sigma(\psi) = 0$

- Según la tabla de verdad en otros casos.

1.1.4. Definiciones de fórmula

Definición: Dos fórmulas son lógicamente equivalentes si todas las valuaciones, ambas entregan el mismo valor de verdad.

Definición: Una fórmula es contradictoria o insatisfacible, cuando para todas las valuaciones σ, la fórmula resulta ser falsa. O sea $\sigma \nvDash \varphi$.

Definición: Una formula es válida cuando todas las valuaciones la hacen verdadera. O sea $\sigma \vDash \varphi$.

Definición: una fórmula es satisfacible, si existe una valuación σ tal que $\sigma \vDash \varphi$.

1.1.5. Consecuencia lógica

Una fórmula φ es consecuencia lógica de un conjunto de fórmulas Σ ssi $\forall\sigma$, $\sigma \vDash \Sigma \supset \sigma \vDash \varphi$.

O sea, si la valuación hace verdadera a la hipótesis, entonces hace verdadera a la fórmula.

1.2. Lógica de predicados

1.2.1. Sintáxis de lógica de predicados

En lógica de predicados, el alfabeto es la unión de:

- Un conjunto P de letras de predicados
- Un conjunto C de símbolos de constante
- Un conjunto F de símbolos de función
- Un conjunto S de símbolos estándar $\{\wedge, \vee, \neg, \supset, \equiv, \forall, \exists\}$
- Un conjunto V de símbolos de variables

1.2.2. Definición de término

- Si $c \in C$, entonces c es un término.
- Si $v \in V$, entonces v es un término.
- Si $t_1, t_2, ..., t_n$ son términos y $f \in F$ es un símolos de función n-ario, entonces $f(t_1, t_2, ..., t_n)$ es un término.
- No hay más términos.

Un término en este lenguaje básicamente es un objeto que puede ser conocido o desconocido. Por ejemplo, una constante por sí sola es un término, las variables también son términos, por que son objetos indeterminados. Cuando en matemática representamos un valor por "x", sabemos que es la representación de un valor, pero no sabemos cual es, de todas formas lo manipulamos y operamos con el.

Finalmente cuando tenemos varios términos y aplicamos un símbolo de función a estos términos, lo que obtenemos es un objeto del dominio, por lo tanto, también es un término. De esta forma tenemos una cantidad infinita de términos.

Ejemplo Supongamos un lenguaje de la aritmética y se define el siguiente lenguaje:

$$P = \{< (\circ, \circ), = (\circ, \circ), \leq (\circ, \circ), par(\circ)\}$$

$$F = \{+(\circ, \circ), *(\circ, \circ), S(\circ)\}$$

$$V = \{w, x, y, z\}$$

$$C = \{0, 1\}$$

¿Cuáles pueden ser términos?

Existe una cantidad infinita de términos, por ejemplo:

- 0 es un término.
- z es un término.

- $S(S(S(0)))$ es un término.
- $+(+(S(0), w), +(0, x))$ es un término.

1.2.3. Definición de fbf

- Si $t_1, ..., t_m$ son términos y p es un símbolo de función m-ario, entonces $p(t_1, ..., t_m)$ es una fbf (átomo).
- Si φ es fbf, entonces $\neg\varphi$ es fbf.
- Si φ y ψ son fbf, entonces $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \supset \psi)$ y $(\varphi \equiv \psi)$ son fbf.
- Si φ es fbf y $v \in V$ entonces $(\forall v)\varphi$ y $(\exists v)\varphi$ son fbf.
- No hay más fbf.

1.2.4. Definición de estructura

Una estructura ε compatible con el lenguaje definido por los conjunto $P = \{p_1, ..., p_n\}$, $C = \{c_1, ..., c_k\}$ y $F = \{f_1, ..., f_m\}$ es una tupla $\langle D, p_1^\varepsilon, ..., p_n^\varepsilon, c_1^\varepsilon, ..., c_k^\varepsilon, f_1^\varepsilon, ..., f_m^\varepsilon\rangle$. En donde:

- D es un dominio (conjunto cualesquiera)
- $c_1^\varepsilon, ..., c_k^\varepsilon \in D$
- Si $p_i \in P$ es un símbolo de predicado n-ario, entonces $p_i^\varepsilon \subseteq D \times D \times ... \times D$ (n veces)
- Si $f_j \in F$ es un simbolo de función m-ario, entonces f_j^ε es una función $f_j^\varepsilon : D \times D \times ... \times D(\text{m veces}) \rightarrow D$

1.2.5. Interpretación

Una interpretación en el lenguaje definido por los conjuntos P, C, F y V es una tupla $\langle \varepsilon, \sigma \rangle$, en donde ε es una estructura compatible con ese lenguaje, y σ es una función de asignación $\sigma : V \to D$

Definición: Interpretación de términos es $I = \langle \varepsilon, \sigma \rangle$

Definición: Un literal es un átomo o átomo negado. Definición: Una clausula es una disyunción de literales.

Ejemplo Sea el siguiente alfabeto para un lenguaje de predicados:

- $P = \{humano(\circ), mortal(\circ), menor(\circ, \circ)\}$
- $C = \{Socrates, Prometeo, Zeus\}$
- $F = \{padre, sqr\}$
- $V = \{x, y, z\}$
- $D = \{1, 2, 3, 4, 5\}$

Se define los símbolos:

$Socrates^{\varepsilon} = 2$

$Prometeo^{\varepsilon} = 4$

$Zeus^{\varepsilon} = 3$

El significado de la función Sqr es:

$Sqr^{\varepsilon} = \{1 \to 3, 2 \to 4, 3 \to 5, 4 \to 1, 5 \to 2\}$

El significado de la función $Padre$ es:

$Padre^{\varepsilon} = \{2 \to 4, 4 \to 3, 3 \to 5, 5 \to 1, 1 \to 2\}$

El significado de los símbolos *humano*, *mortal* y *menor* son:

$humano^{\varepsilon} = \{2\}$

$mortal^{\varepsilon} = \{2, 4\}$

$menor^{\varepsilon} = \{(1, 5)\}$

El significado de la constante es un elemento del dominio, el de las funciones y los símbolos de predicado son un subconjunto del dominio o en este caso que es el conjunto del producto cartesiano del dominio consigo mismo.

$\varepsilon =< D, humano^{\varepsilon}, mortal^{\varepsilon}, menor^{\varepsilon}, Socrates^{\varepsilon}, Prometeo^{\varepsilon}, Zeus^{\varepsilon}, padre^{\varepsilon}, sqr^{\varepsilon} >$

$I =< \varepsilon, \sigma >$

$\sigma = \{x \to 1, y \to 3, z \to 5\}$

Luego es posible interpretar en ε

- $sqr(padre(Socrates))$

  $sqr(padre(Socrates)) = sqr(padre(2)) = sqr(4) = 1$ y en este caso $1 \in D$

Ahora podemos ver que sucede con la expresión:

- $humano(Socrates)$

  $humano(Socrates) = humano(2)$, como en este caso $2 \in \{2\}$, por lo tanto $\varepsilon \models humano(Socrates)$

Un último ejemplo

- $humano(padre(Prometeo))$

  $mortal(padre(Prometeo)) = mortal(padre(4) = mortal(3)$, como en este caso $3 \notin \{2, 4\}$, por lo tanto $\varepsilon \not\models mortal(padre(Prometeo))$

1.3. Resolución

Sistema deductivo de Resolución: Regla de deducción (resolución, version proposicional)

$$\frac{\vdash l_1 \vee l_2 \vee ... \vee l_m \vee p \qquad \vdash l_{m+1} \vee ... \vee l_n \vee \neg p}{\vdash l_1 \vee l_2 \vee ... \vee l_m \vee l_{m+1} \vee ... \vee l_n} \quad (1)$$

En versión de predicados para el sistema deductivo de resolución, se debe considerar la sustitución. Una sustitución σ es una función $\sigma : V \to T$, donde V es el conjunto de variables del lenguaje y T es el conjunto de términos posibles en el lenguaje. Dado un término t o una fórmula φ, $t\sigma$ es el término que se obtiene sustituyendo en t las variables $v \in V$ por syus imágenes $\sigma(v) \in T$. Un unificador θ de dos términos t_1 y t_2 es una sustitución tal que $t_1\theta$ y $t_2\theta$ son sintácticamente iguales. Una sustitución σ_1 es más general que otra σ_2 si existe una sustitución σ_3 que no sea un renombramiento de variables tal que σ_2 es $\sigma_1\sigma_3$ (composición de sustituciones). Una sustitución θ es un MGU (Most General Unifier) de dos términos t_1 y t_2 (de dos átomos a_1 y a_2 si:

- Unifica a t_1 y t_2
- No existe otro unificador de t_1 y t_2 que sea más general que θ.

La regla de resolución de predicados queda dada por:

$$\frac{\vdash l_1 \vee l_2 \vee ... \vee l_m \vee p(t_1, ..., t_n) \qquad \vdash l_{m+1} \vee ... \vee l_n \vee \neg p(t_1^{'}, ..., t_n^{'})}{\vdash (l_1 \vee l_2 \vee ... \vee l_m \vee l_{m+1} \vee ... \vee l_n)\theta} \quad (2)$$

Donde θ es un MGU de $p(t_1, ..., t_n)$ y $p(t'_1, ..., t'_n)$, es decir, de cada t_i y los l_i son literales.

1.4. Prueba Formal

En un programa, cuando se quiere realizar una prueba lógica de éste, basta con establecer un invariante a partir del cual se realizan los reemplazos en la fórmula invariante y se reemplaza con las asignaciones. El código parte con la premisa o condición inicial. Se considera el invariante antes de entrar a partes condicionales de código ya sean if-else o ciclos, y se demuestra que el invariante sigue cumpliendose en ellos, y luego se establece las condiciones de salida, con el invariante más la una sección condicional negada.

```
// asignaciones iniciales

invariante: Axioma

// ciclos o if-else
{
mostrar hacia arriba que el invariante sigue cumpliendose
//Asignaciones
}

mostrar lo que ocurre con el invariante y la condicion de
   entrada del ciclo o if-else
//Asignaciones

mostrar por consecuencia que se cumple la asercion final
```

Ejemplo 1 El siguiente es un algoritmo para calcular la función 2^n en función de n (un entero mayor o igual que cero).

```
exp = 1;
i = 0;
while(i != n)
{
   exp = exp * 2;
   i = i + 1;
}
```

La aserción final es $exp = 2^n$.

Demostrar lógicamente que el algoritmo es correcto, es decir, que la aserción final se satisface al cabo de la ejecución del algoritmo. Un invariante apropiado para el problema es $exp = 2^i$. La precondición es $true$.

A continuación se presenta la solución:

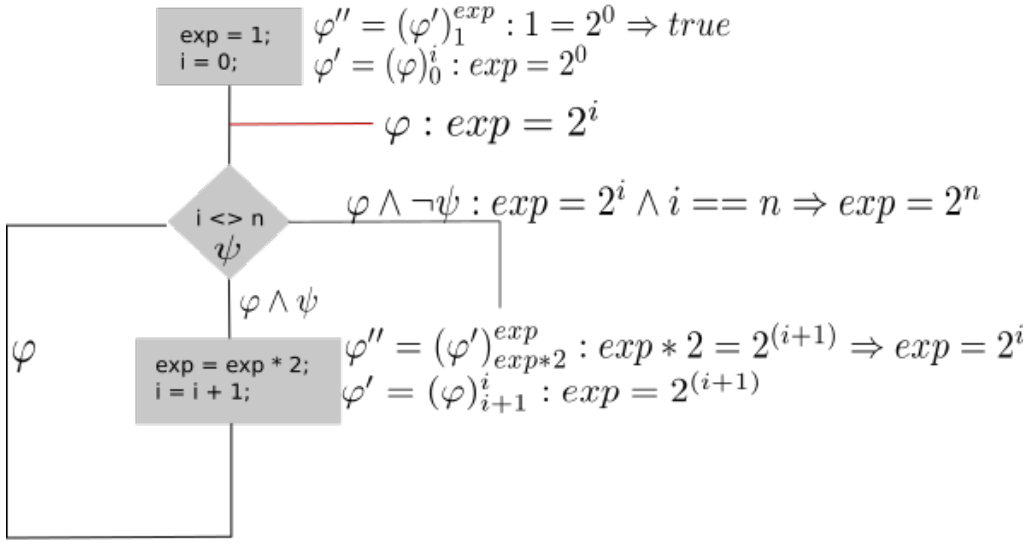

Figura 1: Prueba formal programa para calcular 2^n.

1.5. Ejercicios

1. Considere el siguiente lenguaje

   - $C = \{a\}$
   - $P = \{p\}$
   - $F = \{f\}$
   - $V = \{x, y, z\}$

   Encuentre una interpretación que haga verdadera el siguiente conjunto de fórmulas:

   $$\{p(a), (\forall x)p(x) \Rightarrow p(f(x))\}$$

2. Dados los siguientes axiomas para la suma:

   - $(\forall x)suma(x, 0, x)$
   - $(\forall x, y, z)suma(x, y, z) \Rightarrow suma(x, s(y), s(z))$

   Demuestre usando resolución que $2 + 2 = 4$.

3. Los siguientes axiomas definen (en parte) la suma y el predicado $<$ en los números naturales:

   - $(\forall x)suma(x, 0, x)$
   - $(\forall x, y, z)suma(x, y, z) \Rightarrow suma(x, s(y), s(z))$
   - $(\forall x)menor(0, s(x))$
   - $(\forall x, y)(menor(x, y) \Rightarrow menor(s(x), s(y)))$

   Demuestre usando resolución la siguiente fórmula:

   $$(\forall x)(\exists y)(suma(x, s(s(0)), y) \wedge menor(0, y)$$

4. Probar el siguiente código que pemite restar un entero n a un entero m.

```
x = m;
y = n;

while( y > 0)
{
   x = x - 1;
   y = y - 1;
}
```

Demostrar la correctitud del algoritmo. La precondición para este caso es $n \geq 0$ y la postcondición es $x = m - n$. Utilizar el siguiente invariante: $x - y = m - n \wedge y \geq 0$.

5. El siguiente programa calcula la suma de los elementos de un vector.

```
i = 1;
suma = 0;

while (i != n + 1)
{
   suma = suma + a[i];
   i = i + 1;
}
```

La asercion inicial para este programa es *true*. La postcondición es $suma = \sum_{k=1}^{n} a[i]$. Un invariante adecuado es $suma = \sum_{k=1}^{i-1} a[i]$.

2. Introducción a la programación lógica

En esta parte del apunte, se realizará un pequeño práctico de la programación utilizando Prolog.

2.1. Introducción

Basado en la lógica de primer orden, Prolog utiliza una notación basada en cláusulas de Horn, la cual consta de dos partes:

- La cabeza, que es la conclusión de una implicancia.
- El cuerpo, que consiste en las hipótesis (conjunciones).

La cabeza y el cuerpo se dividen por un el signo : −. A la derecha se escribe la hipótesis y a la izquierda la cabeza o conclusión.

En las hipótesis, puede utilizarse la disyunción, la cual consiste en un punto y coma ";"

2.2. Primer programa en prolog

En lo que sigue, se realiza el primer programa en Prolog, el cual consiste en realizar una serie de afirmaciones (hechos), para establecer el funcionamiento de las operaciones de los booleanos.

La implementación obedece a la especificación utilizando como constructoras las constantes 1 y 0 que representan valores de verdad. divisor La primera operación, es la negación, es una operación unaria que siempre establece el valor de verdad contrario.

La segunda operación es la disyunción, la cual es verdad ssi existe un parámetro verdadero, y cero o falso en caso contrario.

La tercera operación, es la operación conjunción. Esta operación es verdad ssi ambos valores son verdadero y falso en caso contrario.

La cuarta operación, es la implicancia, que tiene un valor verdadero cuando la hipótesis es falsa y falso cuando la hipótesis es verdadera y la implicancia falsa.

La quinta operación es la equivalencia, la cual es verdadera cuando ambos valores de verdad son iguales.

La sexta operación es el orExclusivo, el cual es la negación de la equivalencia.

En lo que sigue, la implementación de las operaciones de los booleanos es realizada.

```
negacion(1,0).
negacion(0,1).
disyuncion(1,X,1).
disyuncion(0,X, X).
conjuncion(1,X,X).
conjuncion(0,X,0).
implicancia(1,X,X).
implicancia(0,X, 1).
equivalencia(1,X, X).
equivalencia(0,X, Z) :- negacion(X, Z).
orExclusivo(1,X, Z) :- negacion(X,Z).
orExclusivo(0, X, X).
```

```
?- orExclusivo(1,1,R).
R = 0.
?- orExclusivo(1,0,R).
R = 1.
?- conjuncion(1,1,R).
R = 1.
?- conjuncion(1,0,R).
R = 0.
?- conjuncion(0,0,R).
R = 0.
```

2.3. El predicado IS

El predicado *is* es empleado para asignar un valor a una variable. En el siguiente código puede apreciarse el uso del predicado *is*.

En la primera línea se establece el hecho que la división entera entre un valor y si mismo es 1 el resto es cero.

La segunda, es una regla que establece que si el valor del dividendo es mayor que el divisor, el cuociente es cero y el resto corresponde al dividendo.

La tercera es una regla que establece que si el dividendo es mayor que el dividendo, entonces la división corresponde al número de veces que el valor del dividendo puede ser restado por el divisor. Del mismo modo, cuando el valor del la diferencia sucesiva es menor al divisor, entonces el valor que queda (valor positivo o cero) corresponde al resto de la división.

```
divisionEntera(X, X, 1, 0).
divisionEntera(X, Y, 0, X) :- X < Y.
divisionEntera(X, Y, D, R) :- Y < X, XT is X - Y, divisionEntera(XT
    , Y, DT, R), D is DT + 1.
```

```
?- divisionEntera(104,3,D,R).
D = 34,
R = 2
```

2.4. Utilización de listas

Es posible realizar operaciones en Prolog, utilizando el tipo abstracto de datos lista. En lo que sigue, se puede apreciar algunos ejemplos de utilizar listas para el procesamiento de datos.

En el siguiente ejemplo, *perteneceElemEnLista*, es un predicado que permite determinar si un elemento pertenece a una lista determinada. Por otra parte, el predicado *contienenElemComunes*, permite determinar, si las listas comparten algún elemento en común. En este caso, la condición para que sean listas disjuntas, no deberían tener ningún elemento en común, por ello el uso de *not*, para negar la condición que contienen elementos comunes.

```
perteneceElemEnLista(X, [X | _]).
perteneceElemEnLista(X, [_ | L]) :- perteneceElemEnLista(X, L).
```

```
listasDisjuntas(L, M) :- not(contienenElemComunes(X, L, M)).
contienenElemComunes(X, L, M) :- perteneceElemEnLista(X, L),
    perteneceElemEnLista(X, M).
```

Un ejemplo de ejecutar el código anterior se presenta a continuación:

```
?- listasDisjuntas([1,2,3], [4,5,6,7,8,9]).
true.

?- listasDisjuntas([1,2,3], [4,5,6,7,8,9,2]).
false.
```

El siguiente, es un ejemplo que permite eliminar todas las ocurrencias de un valor dado como parámetro en una lista determinada. El signo de exclamación establece un corte, cuando se satisface se evalua como verdad y todas las opciones pendientes se eliminan del nodo en el árbol de resolución. Esto significa que la solución encontrada es considerada como suficiente, pero seguirá buscando en los predicados a la derecha del signo de exlamación.

```
eliminarOcurrencias(_,[],[]).
eliminarOcurrencias(X, [X | L], R) :- eliminarOcurrencias(X, L, R),
    !.
eliminarOcurrencias(X, [Y | L1], [Z | L2]) :- eliminarOcurrencias(X
    , Y, Z), eliminarOcurrencias(X, L1, L2), !.
eliminarOcurrencias(_, Atom, Atom).
```

Un ejemplo de ejecutar el código anterior se presenta a continuación:

```
?- eliminarOcurrencias(10,[1,2,10,10,23,13,10,9],R).
R = [1, 2, 23, 13, 9].
```

El siguiente ejemplo, presenta la criba de erastótenes, la cual consiste en listar todos los números primos menores que un valor dado como parámetro. El algoritmo genera una lista con todos los valores que no son múltiplo de los números comenzando por el 2 hasta el valor menor al establecido.

En este caso, se utiliza el comando write para listar la lista resultante luego de ejecutar erastotenes y el parámetro establecido.

```
eliminaMultiplos(_, [], _, []).
eliminaMultiplos(Val, [X | L], X, R) :- !, YS is X + Val,
    eliminaMultiplos(Val, L, YS, R).
eliminaMultiplos(Val, [X | L], Y, [X | R]) :- X < Y, !,
    eliminaMultiplos(Val, L, Y, R).
eliminaMultiplos(Val, [X | L], Y, [X | R]) :- YS is Y + Val,
    eliminaMultiplos(Val, L, YS, R).

generaLista(X, L, C) :- X > 1, !, Y is X - 1, generaLista(Y, [X | L
    ], C).
generaLista(_, C, C).

criba([], _, []).
criba([Val | L], Val, [Val | R]) :- Y is 2*Val, eliminaMultiplos(
    Val, L, Y, CS), criba(CS, PS, R).

erastotenes(N) :- generaLista(N, [], C), criba(C, 2, R), write(R).
```

Un ejemplo de ejecutar el código anterior se presenta a continuación:

```
?- erastotenes(100).
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 ,53, 59,
    61, 67, 71, 73, 79, 83, 89, 97]
true
```

2.5. Ejemplos desarrollados en laboratorio

1. El siguiente programa, permite realizar el cálculo del máximo común divisor de dos números.

```
mcd(X, X, X).
mcd(X, Y, Z) :- X < Y, R is Y - X, mcd(X, R, Z).
mcd(X, Y, Z) :- Y < X, mcd(Y, X, Z).
```

2. El siguiente programa, permite realizar el cálculo de la multiplicación de dos números enteros.

```
multi(X,0, 0).
multi(X,1,X).
multi(X,Y,Z) :- YT is Y - 1, multi(X, YT, R), Z is R + X .
```

3. El siguiente programa, permite calcular la potencia de un número elevando a otro positivo. El programa se realiza como una sucesion de multiplicaciones utilizando para ello una llamada recursiva. El caso base se establece cuando el exponete es cero, y que da como resultado el elemento neutro de la operación multiplicación.

```
potencia(X, 0, 1).
potencia(X, 1, X).
potencia(X, Y, Z) :- YT is Y - 1, potencia(X, YT, R), Z is R *
   X .
```

4. El siguiente programa, permite calcular el producto interno entre dos listas de acuerdo a la siguiente ecuación:

$$\sum_{i=1}^{n} X[i] * Y[i]$$

```
prodInterno(ListaX, ListaY, PI) :- prodInterno(ListaX, ListaY,
   0, PI).
prodInterno([],[], PI, PI) :- !.
prodInterno([X|ListaX], [Y|ListaY], ValTemp, PI) :- Temp is X *
    Y + ValTemp, prodInterno(ListaX, ListaY, Temp, PI).
```

5. El siguiente programa, permite mezclar de forma ordenada dos listas previamente ordenadas.

```
mezclar(ListaX, [], ListaX) :- !.
mezclar([], ListaY, ListaY) :- !.
mezclar([X|ListaX], [Y|ListaY], [X,Y|ListaZ]) :- X=:= Y, !,
    mezclar(ListaX, ListaY, ListaZ).
mezclar([X|ListaX], [Y|ListaY], [X|ListaZ]) :- X < Y, !,
    mezclar(ListaX, [Y|ListaY], ListaZ).
mezclar([X|ListaX], [Y|ListaY], [Y|ListaZ]) :- X > Y, !,
    mezclar([X|ListaX], ListaY, ListaZ).
```

6. El siguiente programa, permite establecer si dos listas don disjuntas.

```
pertenece( X ,[X | _]).
pertenece( X, [_ | L]) :- pertenece(X,L).
disjuntos(L1, L2) :- not(comun(X, L1, L2)).
comun(X, L1, L2) :- pertenece(X, L1), pertenece(X, L2).
```

2.6. Ejercicios propuestos

1. Realizar un programa que permita establecer el valor máximo y mínimo entre dos números naturales.

2. Realizar un programa que permita calcular el resto de la división entera. Para ello defina reglas que le permitan establecer la suma entre dos números.

3. Realizar un programa que permita calcular la función de Ackermann:

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0; \\ A(m-1,1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m-1, A(m,n-1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

4. Establecer las reglas para generar un ordenamiento de los elementos contenidos en una lista.

5. Dada una lista de naturales devuelva la suma de todos los elementos que la componen.

6. Dada una lista que contiene listas, obtener una lista formada por la concatenación las listas.

7. Dada una lista de listas, se obtenga una lista de naturales de forma que cada lista se convierta en su longitud.

8. Transformar una lista de naturales en una lista de booleanos, de forma que los números pares queden con un valor verdadero y los impares en falsos.

9. Elevar al cuadrado todos los elementos de la lista.

10. Una operación que permita recibir la lista y la transforme en una lista cuyos elementos sean los mismos que la lista anterior pero en base b.

3. Especificación Algebraica y por Restricciones

En esta parte del apunte, realizaremos una introducción a dos lenguajes de especificación: Algebraico y Basado en Restricciones.

3.1. Especificación Algebráica

Uno de los problemas normales que tienen los informátocos, es querer hacer sin saber qué hacer!!!. Una especificación en principio, es la clave. En esta parte del curso, vamos a usar el lenguaje lógico para especificar software independiente de la implementación.

Las especificaciones algebraicas son una técnica de especificación que usa un lenguaje formal lógico basado en axiomas y operaciones definidas como constructoras. Es bien sabido, que las especificaciones pueden realizarse a partir de lenguaje natural, se apoyan en el lenguaje semi formal (como notaciones estándares), pero nadie asegura que todas las personas van a entender lo mismo.

Usar los mecanismos necesarios para representar valores usando tipos concretos para desarrollar sus operaciones, requiere un formalismo más allá y que es natural para los ingenieros que han tenido una formación rigurosa y con formalismos matemáticos que avalan un lenguaje universal.

Las ventajas de usar una especificación formal son:

- No ambigüedad para la implementación
- Facilita las pruebas o verificación formal
- Posibilita el desarrollo de implementaciones

La especificación define la semántica de un TAD, pero no condiciona la sintaxis de implementación.

Una especificación algebraica no determina un lenguaje de programación, ni obliga a usar mecanismos concretos.

Una operación, puede tener múltiples implementaciones, ya que puede aparecer como una función, una sobrecarga de un operador o un procedimiento.

Podría darse el caso de que una operación puede implementarse mediante un conjunto de subprogramas, cada uno de los cuales resuelva determinados casos, o al contrario, incluso dos operaciones similares se puedan implementar con un único subprograma con parámetros adicionales para distinguir entre ellas.

Una implementación debe responder a la sintaxis que defina la interfaz y satisfacer la semántica que se establezca por la especificación.

¿cuál es el formato que usaremos para la especificación formal?

El formato que vamos a usar para realizar la especificación algebraica es el siguiente, como vimos en clase.

Especificación nombre de la especificacion

Usa: esp.1, esp2, ..., esp.N

Tipo: tipo1, tipo2, ..., tipoN

Sintaxis:

op1:	parámetros	$\rightarrow$	tipoDev
op2:	parámetros	$\rightarrow$	tipoDev
...			
opn:	parámetros	$\rightarrow$	tipoDev

Variables:

Definición de variables con los tipos que se usarán

Ecuaciones:

Axiomas

Fin

3.2. Ejemplos

Realizar la especificación de los números naturales. Pensemos que necesitamos algo que me permita construir los números naturales. Pensemos que existe una operación

Cero, que simplemente me genera el número cero, y otra operación Sucesor, que me genera el siguiente número.

Con estas dos operaciones podríamos construir todo el conjunto de los naturales.

La especificación, la procederemos a realizar utilizando las operaciones Suma, Producto y Menor.

Especificación: Naturales

Usa: Booleanos

Tipos: nat

Sintaxis:

cero	:		$\rightarrow$	nat
sucesor	:	nat	$\rightarrow$	nat
suma	:	nat nat	$\rightarrow$	nat
producto	:	nat nat	$\rightarrow$	nat
menor	:	nat nat	$\rightarrow$	bool

Variables:

$\forall x, y : nat$

Ecuaciones:

suma(x, cero())	=	x
suma(cero(), x)	=	x
suma(x, sucesor(y))	=	sucesor(suma(x,y))
producto(x, cero())	=	cero()
producto(cero(), x)	=	cero()
producto(x, sucesor(y))	=	suma(x, producto(x,y))
menor(cero(), sucesor(x))	=	true
menor(x, cero())	=	false
menor(sucesor(x), sucesor(y))	=	menor(x,y)

Fin especificación

Cuando realicemos una especificación de un TAD genérico, vamos a representar la inclusión de parámetros que representen a los elementos. Dichos elementos deben aparecer al comienzo de la especificación [Elem]. Por ejemplo, como los conjuntos se componen de elementos, éstos pueden ser de cualquier tipo, se incluye como parámetro con la especificación, del que solo se exige que sea de un tipo.

Especificación: Conjuntos[Elem]

Usa: Booleanod, Naturales

Tipos: conjunto

Sintaxis:

vacio()	:		$\rightarrow$	conjunto
unitario	:	elem	$\rightarrow$	conjunto
union	:	conjunto conjunto	$\rightarrow$	conjunto
interseccion	:	conjunto conjunto	$\rightarrow$	conjunto
diferencia	:	conjunto conjunto	$\rightarrow$	conjunto
incluido	:	conjunto conjunto	$\rightarrow$	bool
pertenece	:	elem conjunto	$\rightarrow$	bool
cardinal	:	conjunto	$\rightarrow$	nat

Variables:

$\forall a, b, c : conjunto$

$\forall x, y : elem$

Ecuaciones:

union(vacio(), a)	=	a
union(union(a,b), b)	=	union(a,b)
union(union(a,b),c)	=	union(union(a,c),b)
interseccion(vacio(), a)	=	vacio()
interseccion(union(a,b),b)	=	b
interseccion(union(a,b),c)	=	union(interseccion(a,c), interseccion(b,c)
diferencia(a,vacio())	=	a
diferencia(a, union(a,b))	=	vacio()
diferencia(union(a,b),c)	=	union(diferencia(a,c), diferencia(b,c)
incluido(a, vacio())	=	false
incluido(a, union(a,b))	=	true
incluido(union(a,b), c)	=	incluido(a,c) $\wedge$ incluido(b,c)
pertenece(x, vacio())	=	false
pertenece(x, unitario(y))	=	x == y
pertenece(x, union(a,b))	=	pertenece(x,a) $\vee$ pertenece(x,b)
cardinal(vacio())	=	0
cardinal(unitario(x))	=	1
cardinal(union(a,b))	=	cardinal(a) + cardinal(b) - cardinal(interseccion(a,b))

Fin especificación

3.3. Ejercicios

1. Realizar la especificación de los números enteros con las siguientes operaciones:

   - Cero
   - Sucesor
   - Predecesor
   - Suma
   - Producto

- Diferencia
- Cambio de signo
- Relación de igualdad
- Relación de orden $\leq$

2. Especificar los números complejos con las siguientes operaciones:

   - Considere como constructor la operación *complejo* que recibe como parámetros dos números enteros (parte real y parte imaginaria)
   - Obtener la parte real
   - Obtener parte imaginaria
   - Suma de complejos
   - Resta de complejos
   - Producto de complejos
   - Conjugado de un complejo
   - Relación de igualdad

3. Especificar el tipo de datos Cola con las siguientes operaciones:

   - Crear una cola vacía
   - añadir un elemento al final de la cola
   - Eliminar el primer elemento de la cola
   - Consultar el primer elemento de la cola
   - Determinar si la cola es vacía
   - Invertir la cola
   - Concatenar dos colas

3.4. Especificaciones de tipos de datos

3.4.1. Implementación Booleanos

En lo que sigue, se presentan algunas implementaciones realizadas a partir de la especificación. Se debe considerar que la implementación responde al cómo, en cambio la especificación responde al qué. Por ende, la implementación podría diferir entre un programador y otro, pero nunca responde algo diferente.

Especificación: Booleanos

Usa:

Tipos: bool

Sintaxis:

verdadero	:		→	bool
falso	:		→	bool
negacion	:	bool	→	bool
disyuncion	:	bool bool	→	bool
conjuncion	:	bool bool	→	bool
implicancia	:	bool bool	→	bool
equivalencia	:	bool bool	→	bool
disyuncionExclusiva	:	bool bool	→	bool

Variables:

$\forall b : bool$

Ecuaciones:

negacion(verdadero)	=	falso
negacion(falso)	=	verdadero
disyuncion(verdadero, b)	=	verdadero
disyuncion(falso, b)	=	b
conjuncion(verdadero, b)	=	b
conjuncion(falso, b)	=	falso
implicancia(verdadero, b)	=	b
implicancia(falso, b)	=	verdadero
equivalencia(verdadero, b)	=	b
equivalencia(falso, b)	=	negacion(b)
disyuncionExclusiva(verdadero, b)	=	negacion(b)
disyuncionExclusiva(falso, b)	=	b

Fin especificación

3.4.2. Arboles Binarios

A continuación se realiza la especificación de un Arbol Binario con elementos parametrizados. La especificación de las Listas, como la vimos en clases, la consideramos ya especificada, al igual que los Booleanos.

Especificación: ArbolesBin[elem]

Usa:Booleanos, Listas[Elem]

Tipos: ab

Sintaxis:

crear	:	$\rightarrow$	ab	
plantar	:	ab, elem, ab	$\rightarrow$	ab
izq	:	ab	$\rightarrow_p$	ab
der	:	ab	$\rightarrow_p$	ab
raiz	:	ab	$\rightarrow_p$	elem
esVacio	:	ab	$\rightarrow_p$	bool
preorden	:	ab	$\rightarrow$	lista

Variables:

$\forall e : elem$

$\forall iz, dr : ab$

Ecuaciones:

izq(crear()) = error

izq(plantar(iz, e, dr)	=	iz
der(crear())	=	error
der(plantar(iz, e, dr))	=	dr
raiz(crear())	=	error
raiz(plantar(iz, e, dr))	=	e
esVacio(crear())	=	verdadero
esVacio(plantar(iz, e, dr))	=	falso
preorden(crear())	=	crearListaVacia()
preorden(plantar(iz, e, dr))	=	concatenar(unitaria(e), concatenar(preorden(iz), preorden(dr)))

Fin especificación

3.5. Introducción OCL

OCL es un lenguaje de modelado con el cual se puede contruir modelos de software. Se define como un lenguaje basado en restricciones y muy atado a UML. A partir de éste, es posible establecer de manera formal las características de las clases de software.

De manera reduccionista, se puede afirmar que OCL se basa en lógica de predicados y teoría de conjuntos. Se define como un lenguaje declarativo, que permite describir de manera abstracta lo que realiza la aplicación a través de un conjunto de expresiones que pueden ser verificadas.

El lenguaje de especificación OCL, es estandarizado por la Object Management Group.

3.6. Sintáxis en OCL

3.6.1. Retorno de un metodo sin efectos

A continuación, vemos el ejemplo de la operación espcificada en defniición del valor de resultado retornado.

```
context Cuenta :: getSaldo() : Real
body : self.saldo
```

3.6.2. Valores de atributos

Podemos especificar el valor inicial de un atributo (init) o el valor de un atributo derivado

```
context Cuenta :: saldo : Real
init : 0.0
```

3.7. Escritura de una expresión OCL compleja

3.7.1. Expresiones compuestas

Para las operaciones lógicas (and, or, xor, not,...), es posible usar if e implica:

- *if* expresion1 then expresion 2, *else* exresion3 endif, si la expresión 1 es verdadera, estonces expresion2 deberia ser verdadera sino expresion 3 deberia ser verdadera,
- expresion1 *implies* expresion2, si la expresion 1 es verdadera, entonces expresion2 debe ser verdadera.

Supongamos la existencia en Persona de un atributo booleano derivado *mayor* y de un atributo booleano *casado*

```
context Persona
inv :
  if edad >= 18 then
        mayor = true
  else
    mayor = false
  endif


context Persona
inv : casado implies mayor
```

3.7.2. Variables

Es posible definir variables para simplificar la escritura de una expresión con la sintaxis:

```
let variable : tipo = expresion1 in expresion2
```

Para calcular un impuesto en una expresión que calcula diferentes formulas que corresponden a tramos de imposición diferentes

```
context Persona
inv:
  let montoImponible : Real = sueldo * 0.8 -- 80% del sueldo
  in
    if(montoImponible >= 1000000) -- tramo de 45%
      then impuesto = montoImponible * 0.45
    else
      if (montoImponible >= 500000) -- tramo al 30%
        then  impuesto = montoImponible * 0.3
      else
        impuesto = montoImponible * 10 -- tramo al 10%
      endif
    endif
```

Es posible entonces definir una variable u operación utilizable en la mayoría de las expresiones con la sintaxis

```
def variable : tipo = expresion
```

Por ejemplo

```
context Persona
def : montoImponible : Real = sueldo*0.8

context Persona
def : edadDentroRango(e: Real) : Boolean = (e >= 0) and (e <= 100)

-- Con la operacion, es posible reescribir el invariante sobre la
   edad
context Persona
inv : edadDentroRango(edad) -- la edad se encuentra entre 0 y 100
```

3.7.3. Tipos de operadores

Los tipos y los principales operadores en OCL son los siguientes:

Tipo	Ejemplo de valores	Operadores
Boolean	true false	and or xor not implies if-the-else-endif ...
Integer	1 -5 2134	* / + - abs ...
Real	1 3.14	* / + - abs() floor() ...
String	"Ser o no ser"	concat() size() substring() ...

Podemos usar el valor de un tipo enumerado por la notación

```
nombreEnum :: valor
```

```
context Persona
inv :
  if edad <= 12 then cat = Categoria::infantil
  else
    if edad <= 18 then cat = categoria::joven
    else
      cat = categoria::edulto
    endif
  endif
```

3.7.4. Explotación de la relación de especialización

Las especificaciones entre clases pueden tomar en cuenta las opoeraciones:

- $oclIsKindOf(type)$: verdadera si el objeto es de tipo *type* o un subtipo
- $oclIsTypeOf(type)$: verdadero si el objeto es de tipo *type*

Por ejemplo, la clase *Piloto* deriva de la clase *Persona*. En el contexto de la clase Piloto self.oclIsKindOf(Piloto) y self.oclIsKindOf(Persona) son verdaderas.

3.7.5. Colecciones

Existen varios subtipos del tipo abstracto Collection:

- Set: Conjunto no ordenada de elementos únicos
- OrderedSet: Conjunto ordenado de elementos únicos
- Bag: Colección no ordenada de elementos con posibilidad de repeticiones
- Sequence: Colección ordenada de elementos con posibilidad de repeticiones

Si el objeto referenciado es una instancia de la clase X y la multiplicidad de la asociación de un lado de esta clase es:

- 1, el tipo resultante es un objeto X
- *, el tipo resultante es un Set(X)
- * con restricción ordenado, el tipo resultante es un OrderedSet(X)

Si componemos mas accesos, el resultado puede ser una bolsa o una secuencia.

3.7.6. Operaciones sobre las colecciones

Las operaciones sobre las colecciones son numerosas y podemos usar la notación

```
colleccion -> operacion
```

Podemos usar la flecha en lugar de la notacion punteada

- size() retorna el tamaño de la coleccion (numero de elementos).
- count(obj) retorna el numero de ocurrencias del objeto obj en la colección.
- sum() retorna la suma de todos los elementos reales o enteros de la coleccion.
- exists(elem: T | unaExpresion) o exists(unaExpresion) retorna verdadero si y solko si al menos un elemento de la coleccion satisface la expresion unaExpresion.

- isEmpty() / notEmpty() recorna verdadero si la colección es o no es vacía.
- includesAll(Coleccion) /excludesAll(Coleccion) retorna verdadero si la colección contiene o no contiene elementos de la colección "Coleccion".
- allInstances() retorna todas las instancias de la clase referenciada.
- forAll(elem: E | unaExpresion) o forAll(unaExpresion) retorna verdadero ssi unaExpresion es verdadera para todos los elementos de la colección.
- union retorna la union de dos colecciones.
- interseccion retorna la intersección de dos colecciones
- select(elem: E | unaExpresion) o select(unaExpresion), retorna los elementos de la colección que verifican unaExpresion.
- reject(elem: E | unaExpresion) o reject(unaExpresion) retorna todos los elementos de la colección excepto los que verifican unaExpresion.
- collect(elem: E | unaExpresion) o collect(unaExpresion) retorna un Bag de valores resultantes de la evaluación unaExpresion aplicada a todos los elementos de la colección.

3.7.7. Casteo

Existen situaciones donde se necesita usar alguna propiedad de un tipo definido en un subtipo. Cuando un objeto es un subtipo de otro, es posible castearlo utilizando la operación $oclAsType(OclType)$

3.7.8. Operaciones de referencia

```
allInstances() : set{C}
```

```
let s1: String = 'hola', s2 : String = 'mundo', i1 : Integer = 10
    in String.allInstances()
-- da como resultado set{'hola', 'mundo'}
```

Operaciones para el tipo oclAny

oclIsInvalid() : Boolean