Introducción a Prolog

1. Sintaxis

La sintaxis de Prolog está basada en los términos. Un término es una constante, variable o un término compuesto.

Constantes: Las constantes pueden ser números o átomos. Un átomo es un identificador que comienza con una *letra minúscula* seguido de una cantidad indeterminada de letras o digitos. La lista vacía "`[]`" también se considera como un átomo.

Variables: Son identificadores que comienzan con una *letra mayúscula* y les sigue una cantidad indeterminada de letras o dígitos. Como puede verse, **Prolog es sensible al caso**. La variable (anónima) "`_`" es una variable especial de un sólo caracter.

Términos compuestos: Representan una relación entre otros términos. Su sintaxis es:

`termino ::= <atomo> ‘‘(’’ <termino`$_1$`> [, <termino`$_2$`>, ..., <termino`$_n$`>] ‘‘)’’`

El átomo que se pone antes del paréntesis es llamado **functor**.

Listas: Son términos compuestos que usan el functor “.”. Se escriben de forma abreviada siguiendo la siguiente sintaxis

```
Lista ::= ‘‘[]’’ |
‘‘[’’ <X_1> [, <X_2>, ..., <X_n> ‘‘|’’ <Lista> ] ‘‘]’’
```

Los programas en Prolog están compuestos de una serie de predicados o cláusulas. Cada predicado está formado de una cabeza y un cuerpo. La sintaxis es

```
predicado ::= <termino> [ ‘‘:-’’ <cuerpo> ] ‘‘.’’
   cuerpo ::= <termino> [ {‘‘,’’|‘‘;’’} <cuerpo> ]
```

Si un predicado no tiene cuerpo, se llama hecho, en otro caso, se llama regla. Por lo general, un predicado se identifica con las funciones de otros lenguajes de programación. De esta forma, a los términos contenidos dentro de los paréntesis de la cabeza se les suele llamar argumentos.

Ejemplos:

- **Átomos:** `a, b, x, socrates`
- **Variables:** `X, _, Socrates`
- **Términos compuestos:** `hombre(socrates), mayorque(5, 3), f(g(a,b), h(x))`
- **Listas:** `[a,b,c], [1, [2,3], hombre(X)], [1|[2,3]], [X|R]`
- **Hechos:**

```
hombre(socrates).
mortal(X).
mayorque(5, 3).
```

- **Reglas:**

```
mortal(X) :- hombre(X).
abuelo(X, Y) :- padre(X, Z), padre(Z, Y).
```

2. Semántica

Los programas (correctos) de Prolog son *argumentos lógicos válidos*. Es decir, se asume que todos los predicados de un programa afirman algo que es verdadero. Aquello que no está contenido en el programa es falso.

Además Prolog tiene una *base de datos de predicados* interna, donde se almacenan los predicados del programa. Durante la ejecución, la *máquina de inferencias* busca los predicados en esta base de datos.

Los átomos representan información básica. Los hechos representan relaciones básicas verdaderas, y las reglas representan afirmaciones condicionales. Cada uno de estos predicados tiene un equivalente lógico:

- Una regla de la forma `P :- Q` equivale a la fórmula lógica $Q \Rightarrow P$.
- El término compuesto `P, Q` equivale a $P \wedge Q$.
- El término compuesto `P; Q` equivale a $P \vee Q$.

Una computación en Prolog es *probar la validez de un argumento*.

Para ejecutar un programa, primero se debe hacer que el *intérprete* de Prolog *consulte* un programa fuente (esta es la operación análoga a la compilación, y de la misma forma, cualquier error en el código fuente es reportado en este momento). Si no se reportan errores, entonces los predicados del programa se cargan en la base de datos interna.

Después, el usuario debe introducir la *meta* en el intérprete de Prolog que desea probar. La meta es un término compuesto o puede ser una serie de conjunciones o disyunciones de metas.

Para probar dicha meta, el intérprete de Prolog consulta su base de datos y aplica una *maquinaria de inferencias* (La Resolución) para verificar si la meta se puede deducir de las premisas (programa) o no. Si la meta se puede deducir, la computación tiene éxito, y sino, entonces se produce un **fallo**. Es decir, el **fallo** es equivalente a la falsedad lógica.

La maquinaria de inferencias funciona de la siguiente forma...

2.1. Intérprete de Prolog

Entrada: Una meta G y un programa P.
Salida: Una instancia de G que es una consecuencia lógica de P, o "no" en caso contrario.
Algoritmo: Inicializar el *resolvente* R a G
mientras $R \neq \varnothing$ **hacer**
Escoger *alguna* meta $A \in R$.
Escoger *alguna* cláusula $A' :- B_1, ..., B_n \in P$, tal que A y A' son unificables con un **mgu** θ
(Si no existe tal cláusula, terminar el ciclo).
Reemplazar A por $B_1, ..., B_n$ en R.
Aplicar θ a R y a G.
Si $R = \varnothing$ entonces poner G en la salida y responder "yes".
en caso contrario, responder "no".

El término "**mgu**" significa *Most General Unifier*, o *Unificador Más General*. Para ver cómo se obtiene, es necesario ver el algoritmo de **Unificación**...

2.2. Unificación

Entrada: Dos términos T_1 y T_2 a ser unificados.
Salida: θ, el **mgu** de T_1 y T_2 o un "`fallo`".
Algoritmo: Inicializar la sustitución $\theta \leftarrow \varnothing$.
Inicializar la pila P con $T_1 = T_2$.
Inicializar `fallo` $\leftarrow$ `falso`.
mientras $P \neq \varnothing$ y $\neg$`fallo` **hacer**
Sacar $X = Y$ de P.
caso

- X es una variable que no ocurre en Y:
  Sustituir Y por X en P y en θ, agregar $X = Y$ a θ.
- Y es una variable que no ocurre en X:
  Sustituir X por Y en P y en θ, agregar $Y = X$ a θ.
- X y Y son variables o constantes idénticas: continuar.
- X es $f(X_1, ..., X_n)$ y Y es $f(Y_1, ..., Y_n)$ para algún *functor* f y $n > 0$:
  Agregar $X_i = Y_i$ para $i = 1, ..., n$ en P.
- En otro caso: `fallo` $\leftarrow$ `verdadero`.

Si `fallo` **entonces** responder "`fallo`" **otro** responder θ.

La máquina de inferencias es **no-determinista** (no se especifica cuál meta se debe escoger del resolvente ni cuál predicado del programa). En la práctica, se toma la siguiente meta del resolvente, y se van escogiendo los predicados tal como aparecen en el programa.

Sean $A_1, ..., A_n$ las metas en el resolvente R, y sean $P_1, ..., P_m$ los predicados del programa actual cargado en la base de datos. Supóngase que las metas $A_1, ..., A_{i-1}$ ya han sido probadas para algún $1 < i < n$ y ahora deseamos probar la meta A_i.

Comenzamos escogiendo el predicado P_1 para satisfacer la meta A_i. Si no se encuentra un **mgu** θ entonces se produce un `fallo` y escogemos P_2 y volvemos a probar. De esta forma, todos los predicados $P_1, .., P_m$ se prueban contra A_i hasta encontrar algún predicado P_j que sea unificable con A_i, o agotar todas las posiblidades. Si existe tal predicado, entonces A_i se satisface y continuamos el algoritmo con A_{i+1}.

Pero si no existe un predicado P_j que satisfaga a A_i, entonces se produce un `fallo`. Este fallo provoca que se haga un retroceso o *backtracking*. Es decir, desechamos A_i y regresamos a A_{i-1}. Como habíamos supuesto que A_{i-1} estaba satisfecho, entonces debe existir un predicado P_k que satisfizo esta meta. De forma que ahora continuamos tratando de satisfacer A_{i-1} *pero a partir del predicado* P_{k+1}. Si todos los predicados han sido agotados, entonces regresamos al nivel anterior y así sucesivamente hasta encontrar una combinación de predicados que satisfagan todas las metas o se intente regresar al nivel 0, en cuyo caso, la meta original no era satisfacible y el programa termina con un `fallo`.

Como puede verse, **el flujo de control en Prolog no es lineal**.

En lugar de ello, *el control pasa de atrás hacia adelante varias veces*, según sea necesario con tal de satisfacer la meta especificada por el usuario. Esto contrasta fuertemente con el estilo tradicional de programación imperativo, donde cada instrucción se ejecuta en secuencia. En la programación lógica, la máquina de inferencias decide cuáles instrucciones se han de ejecutar y en qué orden. El programador sólo debe especificar que meta se debe de probar y el intérprete se encarga del resto.

Por otro lado, del algoritmo anterior, se puede ver que la unificación se representa con el símbolo "=". Debe notarse que la operación más importante de cualquier lenguaje de programación es la asignación. Sin embargo, Prolog *no tiene asignación*. En su lugar, utiliza la unificación para *asignar valores en ambos sentidos de la igualdad*. La unificación no sólo asigna valores, sino que también puede asignar código o *valores libres*.

Algo más que se debe aclarar: en Prolog, las variables pueden ser libres (cuando todavía no tienen valor) o ligadas (cuando ya se les asignó un valor). Una vez que una variable obtiene un valor (a través de la unificación) *no se le puede asignar ningún otro valor*. La única manera en que Prolog libera los valores de las variables es cuando existe un `fallo` y se produce un retroceso o *backtracking*. Al impedir que se asigne más de un valor a una variable se está logrando la transparencia referencial de la programación funcional. Cada ocurrencia de la variable anónima `_` es independiente y esta variable *siempre queda libre*.

Finalmente, la unificación *no es una evaluación aritmética*. En prolog, los operadores aritméticos no son más que casos particulares de términos cuyo functor se escribe en medio. Es decir, la expresión `A + B` es equivalente, en todos los sentidos, al término compuesto `+(A, B)`. Si se desea efectuar una evaluación aritmética se usa el operador especial `is`.

Ejemplos:

1)	2)	3)

`?- A = 3.`
`A = 3`
`yes`
	`?- 3 + 5 = X.`
	`X = 3 + 5`
	`yes`
		`?- X = Y, Y = a.`
		`X = a, Y = a`
		`yes`

En el ejemplo 1, el símbolo `?-` representa el *prompt* de Prolog. La meta a probar es unificar la variable (libre) `A` con el número `3`. El intérprete responde con el **mgu** diciendo simplemente `A = 3` (se asignó un `3` a la variable `A`) y termina diciendo "`yes`" indicando que la meta pudo satisfacerse.

En el ejemplo 2 se tiene una suma. Sin embargo, la unificación entiende la expresión `3 + 5` como el término compuesto `+(3,5)` (para ver que esto es así pruebe la consulta "`?- display(3 + 5).`").

El tercer ejemplo es interesante puesto que muestra lo que pasa cuando se unifican *2 variables libres*. En este caso, las variables se unifican *compartiendo valor*. Cuando a una de las variables se le asigna un valor, la otra de forma instantánea toma el mismo valor. Además, se ha usado la coma para indicar una meta compuesta: deseamos satisfacer ambas metas.

Ejemplos:

```
4) ?- X is 3 + 5.
   X = 8
   yes
```

En el ejemplo 4 se utiliza el operador "`is`" para hacer una evaluación aritmética. Nótese que *en este caso, la variable siempre debe estar del lado izquierdo*.

```
5) ?- f(X,g(a,Y),h(Z)) = f(z,g(Z,b),W).
   X = z, Y = b, Z = a, W = h(a)
   yes
```

El ejemplo 5 muestra una unificación de dos functores. Esto se resuelve unificando de forma recursiva los argumentos de los functores. Los functores deben tener el mismo nombre y la misma cantidad de argumentos para poder unificarse. Además, si algún par de sus argumentos no es unificable, la unificación del functor fallará también.

Finalmente, considere el siguiente ejemplo:

```
6) ?- X = f(X).
   ∞
```

En este ejemplo, se desea unificar una variable contra un término que contiene a esa misma variable. Sin embargo, no existe forma posible de asignar un valor a `X` que satisfaga esta restricción (intente hacerlo). Si se intenta ejecutar esta consulta, el intérprete de Prolog probablemente entrará en un ciclo infinito. Este es el problema de ocurrencia o el *occurs check*. Algunos compiladores de Prolog proveen instrucciones especiales para efectuar la unificación con esta prueba.

Ejercicio

Aplicando los conocimientos adquiridos sobre la máquina de inferencias de Prolog y la unificación, analice el siguiente programa y responda:

- ¿Cuál es la salida si se ejecuta la consulta "`main1(X,Y,Z).`"? ¿Qué pasa si escribe "`;`" en lugar de `<Enter>` después de la primera respuesta?
- ¿Cuál es la salida si se ejecuta la consulta "`main2.`"?
- ¿Cómo se puede modificar el programa para que muestre la salida "`16,17,26,27,36,37,46,47,56,57`"?

```
a(1).
a(2).
b(3).
b(4).
b(5).
c(6).
c(7).
main1(X,Y,Z) :- a(X), b(Y), c(Z).

main2 :- a(X), b(Y), c(Z),
       write(X), write(Y), write(Z), nl, fail.
```

2.3. Escritura de programas en Prolog

Recuerde que todos los programas en Prolog deben ser, al mismo tiempo, afirmaciones válidas. Conviene especificar las *precondiciones* que se deben cumplir antes de ejecutar un predicado y las *postcondiciones* que debe satisfacer el predicado una vez que termine su ejecución. *Cada regla o hecho que forme parte del mismo predicado debe cumplir estas restricciones*.

En términos prácticos, un programa en Prolog es simplemente una colección de hechos y reglas (predicados). Veamos un ejemplo:

```
% padre(Padre, Hijo).
padre(dario, xerxes).
padre(xerxes, artaxerxes).
padre(pedro, juan).
```

Comenzamos definiendo una relación llamada `padre/2` (la notación `functor/N` indica que el `functor` lleva `N` argumentos) que afirma que su primer argumento es "`padre`" de su segundo argumento. Después afirmamos que es cierto que `dario` y `xerxes` pertenecen a esta relación, etc.

Una vez que se ha terminado de escribir los hechos, se escriben las reglas. Deseamos introducir una relación "`abuelo(Abuelo, Nieto)`" que indica que `Abuelo` es abuelo de `Nieto`. Lo hacemos codificando la siguiente regla:

```
% abuelo(Abuelo, Nieto).
abuelo(X, Y) :- padre(X, Z), padre(Z, Y).
```

La regla se puede interpretar de la siguiente forma: "Es cierto que `X` es abuelo de `Y` si sucede que existe un `Z` tal que `X` es padre de `Z` y `Z` es padre de `Y`". Note que no se ha especificado *cómo* se ha de encontrar ese elemento `Z`, pero eso es precisamente lo que Prolog resuelve usando su máquina de inferencias.

Hasta cierto punto, Prolog permite un nivel más alto de abstracción al codificar algoritmos: en lugar de preocuparse por **cómo** hacer cierta tarea, el programador se concentra en **qué** es lo que quiere hacer.

2.4. Manejo de listas

Debido a la sintaxis con la que se escriben las listas en Prolog, sólo es posible hacer operaciones de inserción y eliminación sobre la *cabeza* de una lista. Además, debido a la *transparencia referencial*, sólo es posible efectuar estas operaciones creando una nueva lista al mismo tiempo.

- Inserción: `NLista = [Elem | Lista]`. Esta unificación agrega el elemento `Elem` al principio de la lista `Lista` y crea una nueva lista `NLista` con el resultado.
- Eliminación: `Lista = [ _ | NLista]`. Esta unificación "abre" la lista `Lista` e *ignora* el primer elemento usando la *variable anónima* "`_`". El resultado es una nueva lista `NLista` que tiene el resultado de eliminar la cabeza de `Lista`.

Usando sólo estas operaciones de inserción y eliminación se pueden programar todas las operaciones sobre listas. Por ejemplo, supóngase que queremos codificar una funcion "`miembro(X, Lista)`" que sea verdadero cuando el elemento `X` esté en `Lista`. Podemos codificarlo de la siguiente forma:

```
% miembro( X(i), Lista(i) ).
% Precond :  X y Lista son datos de entrada.
% Postcond:  Se satisface si X ∈ Lista.
miembro( X, [X | _ ] ).
miembro( X, [ _ | R ] ) :-
   miembro(X, R).
```

En la definición de la función se ha puesto una `(i)` indicando que es un parámetro de entrada (debe tener un valor al "llamar" al predicado). El primer hecho afirma simplemente que si `X` aparece en la cabeza de la lista, entonces es cierto que es miembro de esa lista. La regla afirma que, sin importar lo que tenga en la cabeza la lista, `X` es miembro de la lista si es miembro de su *resto*. Visto de otra forma, si el elemento no se encuentra en la cabeza, entonces *lo eliminamos* y repetimos la operación en lo que sobra de la lista.

Se ha diseñado el predicado miembro para aceptar dos datos a la entrada:

```
?- miembro(3, [1,2,3,4,5]).
yes
?- miembro(a, [1,2,3,4,5]).
no
```

¿Pero qué pasa si en lugar de poner datos en la entrada se ponen *variables libres*?

Al hacer esto, estamos de cierta forma violando las precondiciones formales del predicado. Sin embargo, Prolog intentará satisfacer el predicado de todas formas. ¿Qué sucederá si hacemos la siguiente consulta?

```
?- miembro(X, [1,2,3]).
```

...

La consulta produce el siguiente resultado:

```
?- miembro(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
no
```

Esta consulta equivale a la pregunta: ¿Qué es miembro de la lista `[1,2,3]`? La respuesta es: los elementos 1, 2 y 3. Note que para generar todas las respuestas fue necesario presionar "`;`". Además, después de la última respuesta, ya no es posible obtener más soluciones y Prolog termina con un "`no`".

Un predicado que originalmente se diseñó para comprobar la membresía de un elemento en una lista, ahora sirve para recorrer una lista, sin haber modificado una sola línea de código.

Esta es otra de las características de Prolog: un programa puede funcionar para más cosas de las que fue diseñado inicialmente sin que el programador tenga que modificar el código. La única limitante de esto, es lo que permite la máquina de inferencias interna y la unificación.

Ejercicio: Implemente los siguientes predicados sobre listas:

- `pegar(L1, L2, LRes)`: pega las listas `L1` y `L2` y deja el resultado en `LRes`.
- `quita(X, L, LRes)`: elimina la primera ocurrencia de `X` en `L` y deja el resultado en `LRes`.
- `quitaTodos(X, L, LRes)`: elimina todas las ocurrencias de `X` en `L` y deja el resultado en `LRes`.
- `enesimo(N, X, L)`: regresa el `N`-ésimo elemento de `L` en `X`.
- `elimEnesimo(N, L, LRes)`: elimina el `N`-ésimo elemento de `L` y deja el resultado en `LRes`.

- `invierte(L, LRes)`: invierte la lista `L` y deja el resultado en `LRes`.
- `selecciona(X, L, LRes)`: extrae algún elemento `X` de `L` y deja el resto de la lista en `LRes`. Este predicado debe seleccionar *todos* los elementos de `L` cuando suceda el retroceso.
- `min(X, L, LRes)`: extrae el número `X` más pequeño de la lista `L` y deja el resto de la lista en `LRes`.
- `insertaOrd(X, L, LRes)`: inserta el número `X` en la posición adecuada de la lista ordenada `L` y deja el resultado en `LRes`.

Ejercicio: Programe la ordenación de listas por selección, inserción y *quicksort* (en todas las cabeceras mostradas abajo, `L` es la lista a ordenar y `LOrd` es la lista ordenada devuelta por los algoritmos).

- `seleccion(L, LOrd)`: Se extrae el elemento más pequeño de `L` y se pone al final de una lista temporal `Lp`. El algoritmo se repite hasta dejar `L` vacía, en cuyo caso, `Lp` contiene la lista ordenada.
- `insercion(L, LOrd)`: Se toma el siguiente elemento de `L` y se inserta de forma ordenada en una lista temporal (ordenada) `Lp`. El algoritmo continúa hasta que todos los elementos de `L` han sido insertados en `Lp`, en cuyo caso, `Lp` contiene la lista ordenada.
- `quicksort(L, LOrd)`: `L` se parte en dos listas L_- y L_+ de acuerdo a algún elemento pivote `p` de `L` de forma que todos los elementos de L_- son menores (o iguales) que `p` y todos los elementos de L_+ son mayores. Después se ordenan L_- y L_+ usando `quicksort` y se fusionan (el pivote debe de ponerse en medio de ambas listas durante la fusión).

3. La Base de Datos

Para manipular la base de datos de Prolog se usan los siguientes predicados:

- `assert( <Clausula> ).` `assertz( <Clausula> )`. Agregan la `<Clausula>` al final de la base de datos de Prolog.
- `asserta( <Clausula> )`. Agrega la `<Clausula>` al principio de la base de datos.
- `retract( <Clausula> )`. Elimina la primera cláusula unificable con `<Clausula>` de la base de datos.
- `retractall( <Clausula> )`. Elimina todas las cláusulas unificables con `<Clausula>` de la base de datos.

Usando estos predicados es factible hacer que se agregen nuevas cláusulas a la base de datos de Prolog durante la ejecución del programa. Sin embargo, estas cláusulas no sólo son datos o hechos, *también pueden ser reglas enteras*. Esto se debe a que los datos y el código se representan de la misma forma en Prolog, permitiendo así, la metaprogramación...

4. Metaprogramación

Al igual que LISP, Prolog provee de facilidades para crear código ejecutable en tiempo de ejecución. Pero, a diferencia de LISP, Prolog tiene una base de datos. De forma que hacer que un programa crezca o evolucione durante su ejecución es mucho más transparente para el programador.

Además, se cuenta con otra herramienta para formar cualquier predicando en tiempo de ejecución: el operador ***univ***. Este operador se representa con el símbolo "`=..`" y lleva dos argumentos:

`<Predicado> =..  ‘‘[’’ <Functor> [, <Arg`$_1$`>, ..., <Arg`$_n$`>] ‘‘]’’.`

Del lado izquierdo se pone un predicado y del lado derecho, una lista que representa dicho predicado. Este operador *funciona en ambos sentidos*. De forma que puede usarse para crear nuevos predicados o para analizar predicados ya existentes.

A continuación se muestra un fragmento de código que lee el nombre de un predicado de la consola y luego lo ejecuta, usando el operador *univ*:

```
main( Resp ) :-
   read( Termino ),
   Pred =..  [ Termino, Resp ],
   Pred.
```

Puesto que los datos y el código tienen la misma representación en forma de predicado, es posible asignar un predicado a una variable libre y luego ejecutarlo simplemente al “satisfacer” dicha variable. Esto permite hacer construcciones de código genérico como la siguiente:

```
% if_then_else(Cond(i), Then(i), Else(i))
if_then_else(C, T, _) :- C, !, T.
if_then_else(_, _, E) :- E.
```

Finalmente, es posible implementar el polimorfismo en Prolog mediante estos mecanismos. De hecho puede verse que el polimorfismo no es más que un sub-caso de la metaprogramación.

5. Análisis del Lenguaje Natural

Otra ventaja que provee Prolog es el análisis del lenguaje natural. Debido a su naturaleza no-determinista, es posible codificar programas sencillos que busquen de forma automática la estructura correcta de una oración con partes opcionales (cuyas reglas de producción incluyen símbolos terminales vacíos de forma indeterminada). Prolog provee el operador "`-->`" que utiliza **listas diferenciales** para representar la cadena a ser analizada. La gramática resultante es llamada Gramática de Cláusulas Definidas (Definite Clause Grammar).

Ejemplo:

```
s --> [].
s --> [a], s, [b].
```

Usando DCG, los símbolos **no-terminales** se representan con **predicados** de Prolog y los **terminales** con **listas de átomos**. Así pues, la gramática anterior acepta el lenguaje $\mathcal{L} = \{a^n b^n | n \geq 0\}$. Nótese que la aplicación de la regla "`s --> [].`" es no determinsta (en cualquier momento se podría aceptar la cadena vacía)...

Ejercicio:

Haga las reglas de producción para una gramática que acepte el lenguaje $\mathcal{L} = \{a^n b^n c^n | n \geq 0\}$.

Antes de comenzar, cabe preguntarse: ¿Es posible escribir una gramática libre de contexto que acepte dicho lenguaje?

La respuesta es **no**. Sin embargo, eso no impide que se pueda codificar en Prolog. Recuerde que los no-terminales son predicados y pueden llevar argumentos extra (por ejemplo, una variable para contar la cantidad de a's se han sido aceptadas). Además, es posible intercalar código de Prolog en las reglas de producción usando las llaves "$\{\}$ ".

Entonces, teniendo en cuenta estas consideraciones: ¿cómo se vería un programa que aceptara esta gramática?

```
s(0) --> [].
s(N) --> a(N), b(N), c(N).
a(1) --> [a].
a(N) --> [a], a(M), {N is M + 1}.
b(1) --> [b].
b(N) --> [b], b(M), {N is M + 1}.
c(1) --> [c].
c(N) --> [c], c(M), {N is M + 1}.
```

Como se ha agregado un argumento extra, la consulta también se modifica:

```
?- s(N, [a,a,b,b,c,c], []).
N = 2
yes
```

```
?- s(N, [a,a,b,b,c], []).
no
```

Considere el siguiente fragmento de programa:

```
oracion( oracion(Suj, Pred) ) -->
   sujeto( Suj ), predicado( Pred ).

sujeto( sujeto(articulo(N,G,Art), sustantivo(N,G,Sust),
adjetivo(N,G,Adj)) ) -->
   articulo( articulo(N,G,Art) ),
   sustantivo( sustantivo(N,G,Sust) ),
   adjetivo( adjetivo(N,G,Adj) ).

articulo( articulo(sing, masc, el) ) --> [el].
articulo( articulo(sing, fem, la) ) --> [la].
articulo( articulo(plural, masc, los) ) --> [los].
articulo( articulo(plural, fem, las) ) --> [las].
...

adjetivo( adjetivo(sing, X, grande) ) --> [ grande ].
adjetivo( adjetivo(sing, masc, blanco) ) --> [ blanco ].
...
```

Este programa aceptará oraciones de lenguaje natural, pero sólo aquellas donde haya concordancia entre género y número. Además, devuelve una estructura de árbol que representa la oración. (Esto facilita análisis subsecuentes).

6. Operadores

En Prolog, los operadores aritméticos no son mas que predicados donde el functor se escribe infijo. Para ver cómo se representa internamente una expresión aritmética se puede usar el predicado "`display(Expresion)`".

Además, Prolog permite la definición y uso de nuevos operadores mediante la metadirectiva "`:- op( Precedencia, Posicion, Operador )`". Por ejemplo, para definir los operadores lógicos de negación, conjunción, disyunción e implicación se pueden usar las siguientes declaraciones:

```
:- op(500, fy, ~).
:- op(600, xfy, &).
:- op(700, xfy, ;).
:- op(800, xfy, =>).
```

```
?- display(a ; b & ~ c).
';'(a, &(b, ~(c)))
```

Tarea: Implemente un solucionador simbólico de ecuaciones en Prolog. Recuerde que los operadores aritméticos en Prolog **no** producen una evaluación aritmética por sí solos, simplemente son **predicados** que se escriben con el functor infijo.

```
?- solve( 3*a = y - (z + 8) / (x ^ k), x, Sol ).
Sol = x = ((z + 8)/(y - 3*a)) ^ (1/k)
```

Los operadores aceptados deben ser: `+`, `-` (binario y unario), `*`, `/`, y `^` (exponenciación). La incógnita a despejar puede aparecer en cualquier lado de la ecuación, *pero sólo puede aparecer una vez*. La exponenciación genérica en Prolog se implementa con el operador "`**`".