Trabajo práctico final
Programación lógica y funcional

- **Integrantes**: Conti Bruno, Polo Marcos.
- **Profesora**: Pompei Sabrina.

Resumen

En este trabajo hemos seleccionado una serie de problemas, algunos de ellos relevantes para el mundo de las ciencias de la computación, como el de las n reinas, y otros con una orientación más práctica, como el del sistema experto diagnosticador. Luego, basándonos en el paradigma de programación lógica o el funcional, ofrecemos para cada problema una solución.

El contenido del documento está separado en secciones, las cuales tratan distintos temas, por lo que son independientes entre sí y pueden leerse individualmente sin ningún inconveniente. Cada sección toma como punto de partida un problema y hemos tratado siempre de seguir una misma estructura: primero ofrecemos una introducción conceptual, luego desarrollamos una solución y finalmente comprobamos que los resultados sean los esperados.

Tabla de contenidos

Marco teórico

Prolog

PROLOG es un lenguaje de programación perteneciente al paradigma de la Programación Lógica y Declarativa, y su nombre proviene del francés *Programmation Logique*. Su primera implementación data de principios de los años setenta, y tuvo lugar en la Universidad de Marsella (Francia).

Está orientado a la resolución de problemas mediante el cálculo de predicados. Los programas en PROLOG se componen de cláusulas de Horn que constituyen reglas del tipo "modus ponendo ponens".

Actualmente sus aplicaciones son muy amplias, pero principalmente es usado para problemas basados en restricciones (por ej. juegos), la Inteligencia Artificial y los Sistemas Expertos.

Características de PROLOG

- Declarativo: Es un lenguaje declarativo e interpretado, esto quiere decir que el lenguaje se usa para representar conocimientos sobre un determinado dominio y las relaciones entre objetos de ese dominio.
- Lógica de Primer Orden: PROLOG utiliza la Lógica de Predicados de Primer Orden (restringida a cláusulas de Horn) para representar datos y conocimiento.
- Backtracking: PROLOG utiliza un sistema de backtracking para resolver una meta propuesta. Éste consiste en generar un árbol de búsqueda de todas las posibles resoluciones que puede tener la meta en función de la base de conocimientos.

Los programas Prolog están formados por predicados. Los predicados tienen un nombre y aridad (número de parámetros), y pueden estar definidos por una o más cláusulas.

Las cláusulas pueden ser **hechos** (proposiciones que siempre se consideran ciertas) o **reglas**, que son implicaciones lógicas que pueden tener varios antecedentes (las cláusulas), pero un único consecuente.

LISP

Si bien en este trabajo no utilizaremos directamente LISP, sino uno de sus dialectos (Clojure), creemos que vale la pena mencionar algunas características y datos históricos de este lenguaje que ha sido tan influyente, y no solo en el mundo de la programación funcional.

Las ideas para la versión original de LISP (cuyo nombre proviene de LISt Processing) fueron planteadas por John McCarthy alrededor del año 1956; el objetivo era desarrollar un lenguaje de procesamiento de listas de expresiones algebraicas, para aplicarlo principalmente a la investigación en el campo de la inteligencia artificial.

Una de las inspiraciones al crear el lenguaje fue IPL (*Information Processing Language*), en donde se inventó el concepto del procesamiento de listas. Por otra parte, en conexión con un proyecto paralelo de geometría del plano de IBM, parte del equipo implementó rápidamente otro lenguaje de procesamiento de listas en FORTRAN, FLPL (*FORTRAN List Processing Language*), pero agregar soporte a expresiones condicionales y recursividad en este lenguaje requeriría cambios drásticos.

De esta situación nace la primera implementación del lenguaje, que realizó Steve Russell, entre otros, en 1958. Inicialmente se pensaba desarrollar un compilador, pero como era considerado demasiado costoso, se comenzó con la compilación manual en ensamblador de algunas funciones, permitiendo experimentar para obtener buenas convenciones para el enlazado de subrutinas, gestión de la pila y recolección de basura.

A lo largo de los años se desarrollaron numerosos dialectos del lenguaje, entre los cuales más populares están Racket, Scheme, Clojure y Common Lisp.

Algunas de las ideas que fueron implementadas en las primeras versiones de LISP, la mayoría de las cuales fueron innovaciones del lenguaje, son las siguientes:

- Representación de expresiones simbólicas y otra información mediante estructuras de listas en memoria.
- Representación de información en almacenamiento externo mayormente por listas multinivel.
- Composición de funciones para formar funciones más complejas.
- Uso de expresiones-λ (notación de Church) para crear funciones anónimas.
- Diseño iterativo y modificación dinámica de los programas.
- Interpretación condicional de expresiones de conectores booleanos. Por ejemplo, si se está evaluando una expresión OR, se evaluarán sus términos sólo hasta encontrar una expresión verdadera, el resto no.
- Reserva dinámica de memoria.
- Funciones de primera clase. Es decir, pueden tratarse como los demás datos, por ejemplo pasando el objeto de una función como parámetro o usarlas como retorno de otra función.

Problema de las n reinas

Introducción

El problema de las n reinas consiste en asignarle una posición a una cantidad *n* de reinas en un tablero de ajedrez de nxn casilleros, de manera tal que no se ataquen entre sí, esto es, que dos o más reinas no ocupen simultáneamente las mismas filas, columnas o diagonales. Sus orígenes se remontan al año 1848, y fue estudiado por personalidades como Gauss.

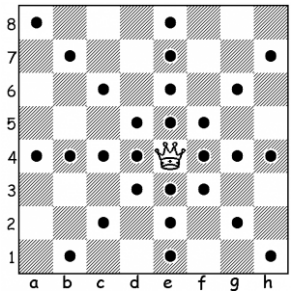


Fig. 1. Movimientos permitidos para la reina en el ajedrez.

El problema suele presentarse en dos formas: la primera consiste en encontrar *una* solución para un *n* dado, mientras que la segunda —de mayor costo computacional— busca encontrar *todas* las posibles soluciones para cierto *n*.

Pertenece a la categoría de problemas NP completos, e incluso se ofrece una recompensa de un millón de dólares para que el que pueda plantear una solución que trabaje en tiempo polinómico, ya que la misma podría derivarse para un sinnúmero de problemas pertenecientes a la misma categoría, además de conducir a una prueba de que P = NP.

Otra característica a destacar del problema es que a medida que *n* incrementa linealmente, el número de soluciones incrementa exponencialmente. Esto puede concluirse, al menos empíricamente, al analizar la Tabla 1.

n	*R(n)*
5	10
6	4
7	40
8	92
9	352
10	724
11	2.680
12	14.200
13	73.712
14	365.596
15	2.279.184

Tabla 1. Cantidad de soluciones para 6 <= *n* <= 15.

Formato de la solución

A la hora de representar la solución, tal vez lo más intuitivo es pensar en un sistema de coordenadas. Podríamos decir, por ejemplo, que la reina 1 estará ubicada en la posición (3, 4) del tablero.

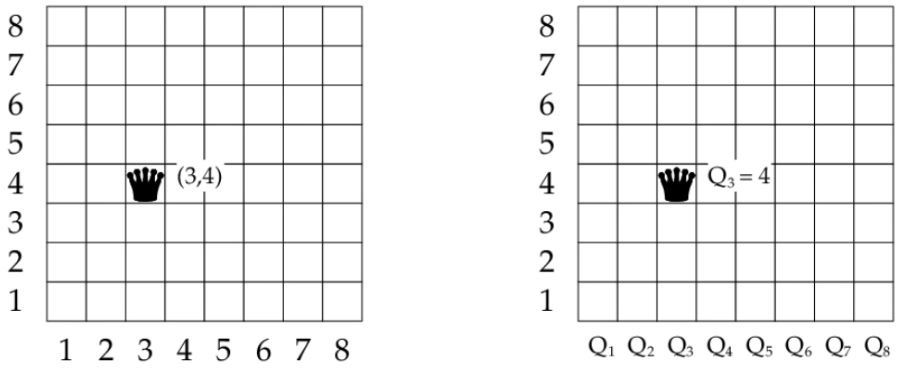


Fig. 2. Dos maneras distintas de representar la solución.

Pero al saber que en cada columna solamente es posible tener una única reina (caso contrario se atacarían), podemos asignar de antemano una columna a cada reina, y al mismo tiempo recurrir a una notación más compacta, en la cual el número de la reina se corresponda con el de la columna. Así, si expresamos $R_3 = 4$, estamos diciendo que la reina de la columna 3 está en la fila 4. De esta forma **reducimos considerablemente el espacio de búsqueda**, ya que como las reinas tienen asignada de antemano una columna, restaría únicamente encontrar las filas en las que estén seguras.

Entonces nuestra solución se representará como una lista con la forma $R_1, R_2, ..., R_n$, donde R_i es una variable que contiene la fila en la que está la reina, mientras que i será la columna. Por ejemplo, al consultar el programa con las soluciones para n = 8, una de ellas es [1,5,8,6,3,7,2,4]. Si visualizamos esto en el tablero esto tendríamos:

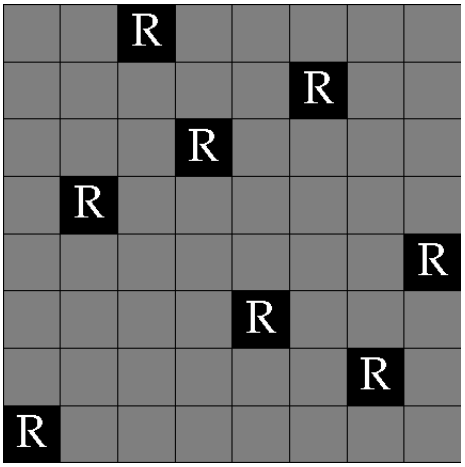


Fig. 3. Visualización de una solución a modo de ejemplo.

Solución propuesta

Describiremos ahora el funcionamiento del programa, comenzando por el momento en el que el usuario ingresa su consulta. Para esto necesitaremos llamar al predicado `n_reinas/2`, que recibirá los parámetros:

- `N`, que representa la cantidad de reinas (que, recordemos, se corresponde con las dimensiones del tablero).
- Rs, que será una lista con las filas ocupadas por las reinas.

Estos dos parámetros, como en todo programa PROLOG, pueden estar definidos o no. Si ambos están definidos, el algoritmo tratará de obtener el valor de verdad de la proposición que le fue suministrada. Y si alguno o ambos están parcial o completamente indefinidos, generará los valores para los que se cumplen las condiciones que hemos definido.

Además de llamar a `n_reinas`, agregamos el parámetro `labeling/2`, perteneciente a una de las librerías que importaremos, ya que gracias al mismo podemos visualizar la solución en un formato entendible para nosotros.

```
?- n_reinas(8, Rs), labeling([ff], Rs).
   Rs = [1,5,8,6,3,7,2,4]
;  Rs = [1,6,8,3,7,4,2,5]
;  ...
```

Fig. 4. Ejemplo de consulta para n = 8.

Pasamos entonces a describir el código en sí, el cual puede encontrarse en texto plano aquí.

```
:- use_module(library(clpz)).
:- use_module(library(lists)).
:- use_module(library(format)).
:- use_module(library(dcgs)).
```

Primero importaremos algunas librerías, para no tener que implementar a mano muchas operaciones elementales. Por ej. CLPZ ofrece predicados para diferencias (`#\=`) o igualdades (`#=`). En las demás librerías tenemos otros como `length/2`, que sirve para comparar la longitud de una lista con un entero, o también `abs/1`, que sirve para calcular el valor absoluto de una expresión.

Luego `n_reinas`, el punto de entrada del programa, se definirá de la siguiente manera:

```
n_reinas(N, Rs) :-
   length(Rs, N),          % (1)
   Rs ins 1..N,            % (2)
   reinas_seguras(Rs).     % (3)
```

(1) El problema establece que la cantidad de reinas (o sea la cantidad de elementos de la lista Rs) debe ser igual a N (o sea las dimensiones del tablero). Entonces verificamos esto con el predicado `length`.

(2) Utilizamos `ins` para asegurarnos de que los valores de la lista de reinas estén entre 1 y N. Como se explicó anteriormente, los valores de `Rs` se corresponden con los de las filas ocupadas por las reinas, por lo tanto no pueden ser menores a 1 ni mayores a N.

(3) Por último tenemos el predicado `reinas_seguras/1`, que como veremos se encarga de comprobar que las reinas no se ataquen entre sí.

Las dos primeras condiciones nos permiten validar que la entrada suministrada al programa sea correcta en términos de la definición del problema. Es recién en la tercera donde hallamos el núcleo del programa.

Continuamos con el predicado `reinas_seguras`.

```
reinas_seguras([]).
```

En este primer predicado definimos que si no hay reinas, entonces es cierto que las mismas están seguras. Este será nuestro punto de corte, ya que a medida que el algoritmo avance irá descartando las reinas que fueron verificadas, hasta finalmente —en caso de dar con una solución— quedarse sin reinas para comprobar .

```
reinas_seguras([R|Rs]) :-
   reinas_seguras_aux(Rs, R, 1),
   reinas_seguras(Rs).
```

La segunda cláusula del predicado `reinas_seguras/1` está preparado para recibir la lista de reinas, la cual será separada en su primer elemento o cabeza (`R`) y el resto de la lista o cola (`Rs`).

En (1) invocamos al predicado `reinas_seguras_aux`, que es donde efectivamente se comprueba que una reina `R` no amenaza o se ve amenazada por ninguna de las otras reinas de `Rs`. Este predicado recibirá tres parámetros:

- La cola de la lista de reinas actual (`R`).
- La cabeza de la lista de reinas actual (`Rs`).
- El tercer parámetro se utiliza para validar que las reinas no se ataquen diagonalmente. Representa la distancia entre las columnas de `R` y la cabeza de `Rs`, de ahí que su valor sea en esta primera llamada 1. Luego irá incrementando de a

uno a medida que el predicado `reinas_seguras_aux` comience a llamarse recursivamente para comprobar las siguientes reinas de `Rs`.

Y en (2) llamamos recursivamente a `reinas_seguras`, suministrando como parámetro solamente la cola de la lista, para que la comprobación realizada en (1) se haga también para todas las subsecuentes reinas. Es aquí donde se define el bucle principal del programa.

Pasamos ahora a describir el predicado `reinas_seguras_aux`, que como describimos recibe en primer lugar una lista con reinas, en segundo lugar una reina en específico y en tercer lugar la distancia entre las columnas de la reina que estamos analizando y la cabeza de `Rs`.

```
reinas_seguras_aux([], _, _).
```

Esta será la condición de corte, el cual solamente será verdadero cuando encontremos que la reina en cuestión no ataca a ninguna de las demás, o sea, cuando lleguemos a la lista de reinas vacías porque ya no nos queda ninguna por verificar.

```
reinas_seguras_aux([R|Rs], R0, D0) :-
   R #\= R0,
   abs(R0 - R) #\= D0,
   D1 #= D0 + 1,
   reinas_seguras_aux(Rs, R0, D1).
```

(1) Como R y R0 contienen las filas en la que están ubicadas las reinas, lo que estamos exigiendo aquí es que las reinas no ocupen la misma fila. Esto invalidaría automáticamente la solución.

(2) En esta condición verificamos que las reinas no comparten una misma diagonal. Para ello, basta obtener el valor absoluto de la diferencia entre las filas y comprobar que este valor sea distinto a la distancia (`D0`). Para entender mejor esto puede verse la Figura 5. Si `R = 2` y `R0 = 5`, tendremos que el valor absoluto de su diferencia será `3`. Gracias al parámetro `D0` también conocemos la distancia entre estas dos reinas, que es de `3`. Entonces como `abs(R0 - R) = D0`, tenemos que en este caso las reinas se están atacando.

Además, el valor absoluto nos permite comprobar el ataque diagonal tanto «desde arriba» como «desde abajo», caso contrario necesitaríamos dos condiciones, una para `R0 - R` y otra para `R - R0`.

(3) Incrementamos la distancia en 1, ya que en el siguiente ciclo vamos a realizar la comprobación sobre la siguiente reina.

(4) Volvemos a llamar `reinas_seguras_aux/3`, descartando la cabeza de la lista actual y enviando la cola, la reina que estamos comprobando y la distancia actualizada.

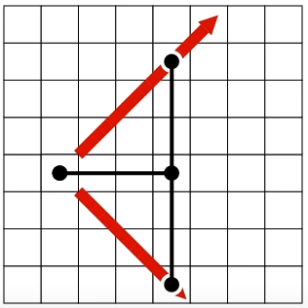


Fig. 5. Comprobación de ataque diagonal

Pruebas finales

Finalizada la implementación, comprobaremos ahora que su funcionamiento es correcto. Comenzamos con una consulta general, sin definir ningún parámetro. Recordamos que el predicado `labeling/2` es para que la lista de reinas se muestre en un formato entendible para nosotros, caso contrario veríamos las expresiones internas de la librería CLPZ.

```
?- n_reinas(N, Rs), labeling([ff], Rs).
   N = 0, Rs = []
;  N = 1, Rs = [1]
;  N = 4, Rs = [2,4,1,3]
;  N = 4, Rs = [3,1,4,2]
;  N = 5, Rs = [1,3,5,2,4]
;  N = 5, Rs = [1,4,2,5,3]
;  N = 5, Rs = [2,4,1,3,5]
;  N = 5, Rs = [2,5,3,1,4]
;  N = 5, Rs = [3,1,4,2,5]
;  N = 5, Rs = [3,5,2,4,1]
;  N = 5, Rs = [4,1,3,5,2]
;  N = 5, Rs = [4,2,5,3,1]
;  N = 5, Rs = [5,2,4,1,3]
;  N = 5, Rs = [5,3,1,4,2]
;  N = 6, Rs = [2,4,6,1,3,5]
;  N = 6, Rs = [3,6,2,5,1,4]
;  N = 6, Rs = [4,1,5,2,6,3]
;  N = 6, Rs = [5,3,1,6,4,2]
;  N = 7, Rs = [1,3,5,7,2,4,6]
;  N = 7, Rs = [1,4,7,3,6,2,5]
;  ...
```

Nótese que para los casos en los que $n = 2$, $n = 3$ o $n = 4$ el programa no ofrece soluciones, lo cual es correcto, ya que para estos valores efectivamente no existe ninguna solución posible. Además, si contamos la cantidad de soluciones para los casos de $n = 5$ o $n = 6$, tenemos que coinciden con los de la tabla definida en la introducción, cuya validez está demostrada matemáticamente.

Pasamos ahora a probar la terminación del programa, agregando `false` a la consulta.

```
?- n_reinas(N, Rs), false.
```

Al ingresar esta consulta el programa nunca devuelve una respuesta, lo cual se corresponde con lo que sabemos del problema, ya siempre habrá soluciones para cualquier n sin importar lo grande que sea. Es por este motivo que nunca llega a devolver nada: siempre hay un n más grande que el anterior para comprobar.

Comprobamos ahora la terminación para un caso discreto, el de $n = 8$.

```
?- n_reinas(8, Rs), false.
false.
```

En esta ocasión obtenemos `false` como respuesta, lo cual también tiene sentido, ya que al ingresar para un n específico vamos a tener un número limitado de soluciones. Lo mismo ocurre para un valor de mayor tamaño, por ej. $n = 100$:

```
?- n_reinas(100, Rs), false.
false.
```

Por último, podemos probar el programa suministrándole una lista de reinas:

```
?- n_reinas(N, [2,4,1,3]).
   N = 4
;  ...
?- n_reinas(N, [2,4,C,D]).
   N = 4, C = 1, D = 3
;  ...
```

En la primera consulta nos informa que esa configuración es válida para n = 4, lo cual efectivamente es correcto. En la segunda consulta le ofrecemos al programa una lista parcial y le pedimos que la complete, lo cual también logra llevar a cabo con éxito.

Como dato extra, el programa no es capaz de generar una respuesta en un tiempo aceptable cuando n es lo suficientemente grande. Esto se debe a la naturaleza del problema, que recordemos está en la categoría de los NP completos. Superado cierto tamaño de n, la solución sencillamente no es capaz de adaptarse a la forma en la que escala la complejidad del problema, independientemente del hardware que utilicemos o las optimizaciones que podamos hacer sobre los algoritmos. Este problema podría verse mitigado al menos un poco si utilizáramos una solución basada por ej. en algoritmos genéticos o estocásticos, pero a fin de cuentas ocurriría lo mismo.

Referencias

- The Power of Prolog, Markus Triska.
- Solving N-queens with Prolog, Markus Triska.
- El problema de las N Reinas, Andreas Spading y Pablo Itaim Ananias, Universidad de Valparaíso.

Sistema experto diagnosticador

Introducción

En la actualidad, es muy notable el avance que han tenido los sistemas y la informática. Muchas de las ramas de estas disciplinas han sido ampliamente investigadas, pero también hay otras en las que falta mucho trabajo por hacer. Entre estas áreas está la Programación Lógica y la Inteligencia Artificial, entre cuyas objetivos nos encontramos el hacer «razonar» a las máquinas, esto es, poder almacenar ciertos conocimientos en una memoria de algún tipo, y obtener, programáticamente, una deducción a partir de ellos.

Una de las banderas de este campo son los **Sistemas Expertos** (SE), los cuales son la representación informática de un experto humano en cierto campo. A partir de sus experiencias propias, estos profesionales han adquirido con los años un nivel de conocimiento, el cual es difícil de adquirir por una persona común en poco tiempo.

Es aquí donde entran los sistemas expertos, ya que pueden ayudar a las personas a obtener algo muy cercano a un consejo profesional pero mediante una computadora. La aplicación de los SE es amplia, desde la matemática con la resolución de teoremas hasta la medicina, con el apoyo a los diagnósticos.

En el desarrollo de este proyecto implementaremos un SE que ayude a las personas a diagnosticar las enfermedades de los peces goldfish. Mediante el SE podrán conocer la enfermedad que afecta al pez y obtener también un tratamiento, y puede ser utilizado por cualquier persona que cuente con esta clase de mascota. Claro está, nuestro SE es solo un prototipo y no es una alternativa ante el trabajo de un profesional.

Reseña general y tecnologías utilizadas

La idea de esta propuesta es crear un sistema al cual describiremos como experto, ya que simulará el rol de un profesional humano que se desempeñe en el área de la medicina para animales, específicamente las enfermedades de peces.

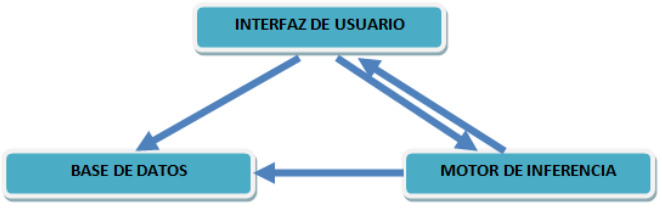


Fig. 1. Los tres componentes de la solución

El diseño de la solución está separado en tres partes claramente definidas (ver Figura 1).

- La interfaz de usuario, a la cual no le dedicaremos mucho tiempo a la hora de analizar el código ya que no es puntualmente lo que nos interesa.
- La base de datos o conocimientos, donde están almacenadas las enfermedades, sus síntomas y tratamientos. Este componente se limita a responder a las consultas de los demás, nunca inicia ningún evento.
- Y por último el motor de inferencia, que se encargará de cuestionar al usuario mediante la interfaz, consultar la base de hechos y determinar el tipo de enfermedad que sufre el pez.

Y las tecnologías utilizadas para el proyecto son:

- Lenguaje de Programación Prolog (IDE SWI-Prolog).
- Librería XPCE para la generación de interfaces de usuario.

Base de conocimientos

A continuación definimos la base de conocimientos que utilizará la solución. Las reglas están compuestas por dos partes: primero el nombre de una enfermedad, y segundo los síntomas que debe presentar el pez para determinar si sufre esa enfermedad o no.

```
conocimiento('hidropesía',
['el pez tiene las escamas levantadas','el pez tiene los ojos sobresalidos',
'el pez tiene falta de apetito','el pez tiene el vientre hinchado']).
conocimiento('vejiga_natatoria',
['el pez tiene el vientre hinchado','el pez tiene problemas de equilibrio',
'el pez tiene falta de apetito','el pez tiene aletargamiento']).
conocimiento('punto_blanco_ich',
['el pez tiene puntos blancos a lo largo del cuerpo y aletas',
'el pez tiene aletargamiento', 'el pez tiene las aletas retraidas']).
conocimiento('estres',
['el pez tiene estados de agresividad', 'el pez tiene falta de apetito',
 'el pez tiene aletargamiento','el pez tiene las venas rojizas y dilatadas']).
conocimiento('parasito_hexamita',
['el pez tiene un hoyo en la cabeza', 'el pez tiene falta de apetito',
 'el pez tiene aletargamiento', 'el pez tiene la cabeza con sangre y tejido muerto']).
```

Motor de inferencia

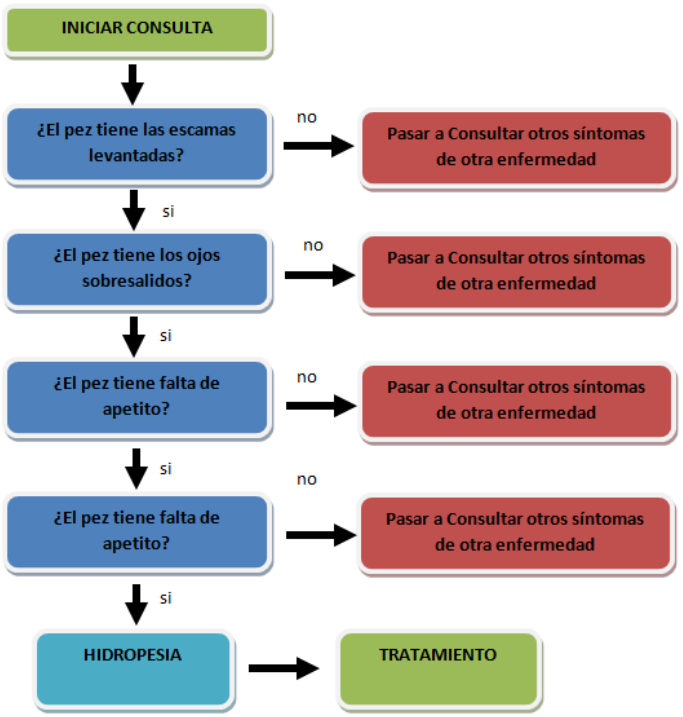


Fig. 2. Lógica del motor de inferencia para el caso de la hidropesía.

Esta parte del sistema se encarga de decidir cuál será el diagnóstico a partir de las preguntas realizadas al usuario. Si bien el código es algo extenso debido al manejo de las interfaces de usuario, el núcleo de la lógica es relativamente simple y puede resumirse de la siguiente manera (ver Figura 2 para un ejemplo específico):

1. El programa toma una enfermedad y comienza a preguntar al usuario si el pez sufre los síntomas que le corresponden a ésta, según están definidos en la base de conocimientos.

    1.1. Si el usuario responde que sí ante un síntoma, el programa continúa consultando por los demás de esa misma enfermedad.

    1.2. Si el usuario responde que no, se pasa a la siguiente enfermedad y se comienza a preguntar de nuevo por los síntomas de la misma, volviendo al punto 2.1. Si las enfermedades comparten síntomas y hay uno o más que ya fueron respondidos, el programa los recuerda y no vuelve a preguntar sobre esos síntomas en específico.

2. El programa finaliza cuando se haya respondido afirmativamente a todos los síntomas de cierta enfermedad, o cuando nos hayamos quedado sin enfermedades que comprobar en nuestra base de hechos.

Solución propuesta

Pasamos a comentar el código de la implementación. Como mencionamos antes, nos concentraremos en el motor de inferencia que es donde encontramos el núcleo del programa.

```
:- dynamic sintomas_conocidos/1.
```

Mediante el predicado dynamic creamos, justamente, un predicado dinámico. Lo utilizaremos para almacenar en tiempo de ejecución las respuestas de los usuarios respecto a los síntomas, ya que este es un conocimiento que no es estático — el valor de verdad de los síntomas puede variar, es decir, en un momento dado un pez puede presentar unos síntomas y otros no.

```
% E: enfermedad
mostrar_diagnostico(E) :- diagnosticar(E), reiniciar_sintomas.
mostrar_diagnostico(diagnostico_desconocido) :- reiniciar_sintomas.
```

Este es el predicado que se invoca cuando hacemos clic en el botón 'Iniciar consulta'. En su primera cláusula, tratará de diagnosticar la enfermedad que sufre el pez mediante el predicado `diagnosticar`. Si esto se cumple, o sea si puede diagnosticar una enfermedad, informará de la misma a la interfaz y luego borra todo lo guardado en el predicado `sintomas_conocidos` mediante `reiniciar_sintomas`. La finalidad de esto es sencillamente que si se inicia una consulta nueva, se haga con la memoria limpia, sin recordar lo que fue respondido anteriormente.

En caso de no poder diagnosticar la enfermedad se continúa con la segunda cláusula, en la cual también limpiamos `sintomas_conocidos`, pero esta vez enviamos a la interfaz la cadena `diagnostico_desconocido`. La interfaz está preparada de manera tal que cuando recibe este valor le indicará al usuario que no se pudo diagnosticar la enfermedad, y que en consecuencia no se puede recomendar un tratamiento.

```
% LS: lista de síntomas
diagnosticar(E) :-
  conocimiento(E, LS),
  demostrar(E, LS).
```

Para poder diagnosticar la enfermedad, lo primero que hacemos es consultar `conocimiento`, nuestra base de conocimientos, para «iterar» sobre las enfermedades y sus síntomas. A su vez, pasamos esto como parámetros al predicado `demostrar`, que es donde determinaremos si el pez sufre una de una enfermedades o no a través de las preguntas que responda el usuario.

```
% S: síntoma, RS: resto de síntomas
demostrar(E, []).
demostrar(E, [S | RS]) :-
  sufre_sintoma(E, S),
  demostrar(E, RS).
```

`demostrar` se llama recursivamente a sí misma, y en cada iteración preguntará al usuario por cada síntoma de la enfermedad. Si llegamos a la lista vacía de síntomas, o sea si no hay más síntomas por los que preguntar, se determina que el pez efectivamente sufre de la enfermedad `E`. En ese momento el programa hace el backtracking para devolver lo obtenido a la interfaz.

```
% R: respuesta
sufre_sintoma(E, S) :- sintomas_conocidos(S).
sufre_sintoma(E, S) :-
  not(sintomas_conocidos(is_false(S))),
  pregunta_sobre(E, S, R),
  R = 'si'.
```

Como su nombre lo indica, el propósito de este predicado es determinar si el pez sufre un síntoma. Para esto puede hacer dos cosas:

1. Consulta `sintomas_conocidos`, que es donde se almacena lo que ya fue respondido por el usuario. Si no encuentra nada (obtiene falso), pasa a la siguiente cláusula.
2. Genera nuevamente una consulta en `sintomas_conocidos`, pero esta vez para asegurarse de que el usuario no respondió negativamente a la pregunta. Si esto es así, se abre un diálogo mediante el predicado `pregunta_sobre`, donde obtendremos la respuesta `R` (sí/no).

```
pregunta_sobre(E, S, R) :-
  preguntar(S, R),
  guardar(E, S, R).

guardar(E, S, si) :-
  asserta(sintomas_conocidos(S)).
guardar(E, S, no) :-
  asserta(sintomas_conocidos(is_false(S))).
```

Con `preguntar` se invoca al diálogo donde el usuario responderá sobre el síntoma. La respuesta será devuelta en la variable `R`. Esta a su vez es enviada al predicado `guardar`. Mediante este llamamos a `asserta`, y es aquí donde guardamos la respuesta (ya sea «sí» o «no») del usuario de forma dinámica durante esta sesión.

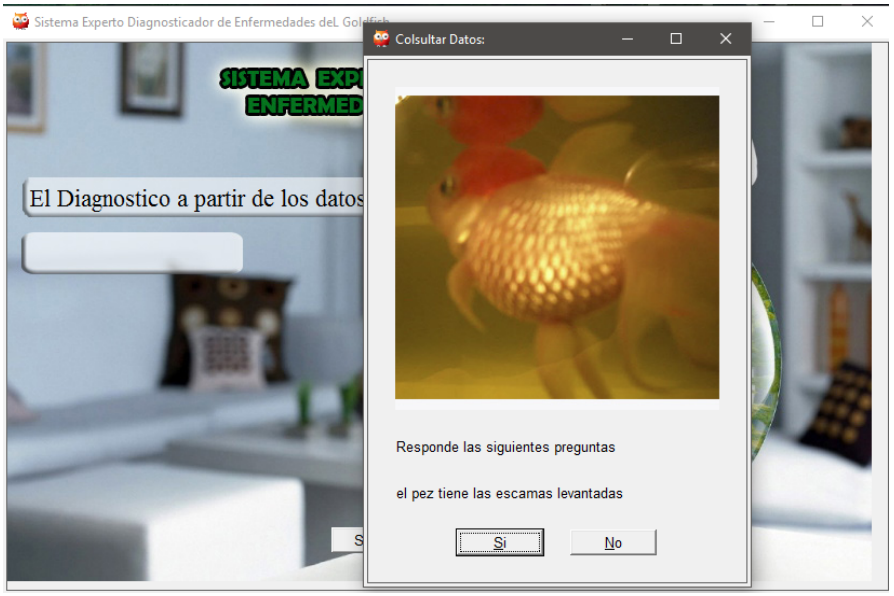
```
reiniciar_sintomas :- retract(sintomas_conocidos(S)), fail.
reiniciar_sintomas.
```

Finalmente, una vez que el programa termina de diagnosticar la enfermedad (positiva o negativamente), se invoca a `reiniciar_sintomas` para eliminar todo lo que guardamos en `sintomas_conocidos` en caso de que sea necesario iniciar una nueva consulta.

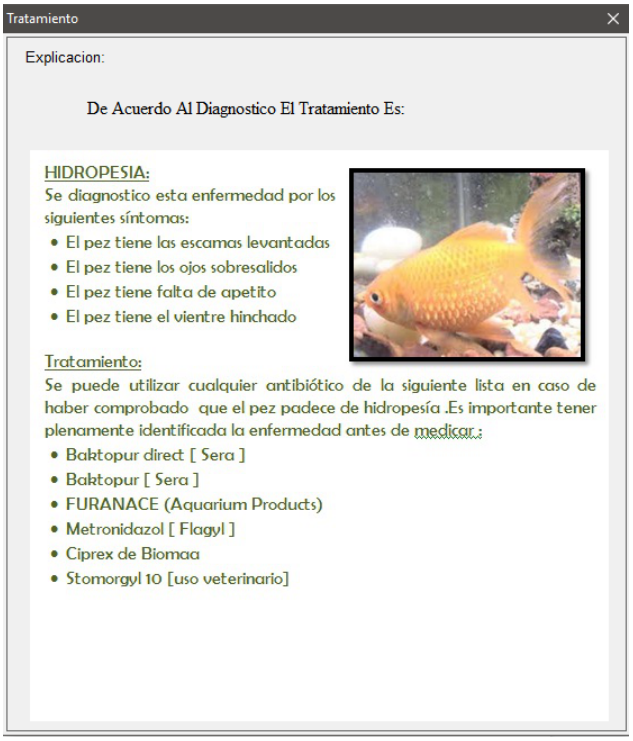
Pruebas finales

Pasaremos ahora a probar el funcionamiento del programa. A continuación tenemos la pantalla de bienvenida del sistema con algunas modificaciones visuales que adaptamos y que se inicia automáticamente al consultar el archivo del sistema.
Nos pareció importante aclarar que no quisimos agregar el código de las vistas en el trabajo práctico para no hacerlo tan largo y tedioso lo cual decidimos centrarnos más en la lógica y el funcionamiento del mismo, de igual forma al final pondremos la fuente del mismo por si desear ver como es la implementación.

Al hacer clic en ‘Comenzar’ se abre la pantalla de diagnóstico. Si luego entramos en ‘Iniciar consulta’ vemos cómo se inicia el proceso de diagnóstico a partir de diálogos donde se van preguntando los síntomas.



Una vez finalizado el diagnóstico, el sistema muestra la enfermedad deducida, o avisa de que falló el diagnóstico en caso contrario. Si se encontró una enfermedad es posible utilizar el botón 'Detalles y Tratamiento' marcado en la imagen para obtener una breve descripción de la enfermedad y un posible tratamiento para la misma.



Conclusiones

Si bien este SE es muy limitado (la base de conocimientos, por ejemplo, podría expandirse mucho más), nos permite entender un poco el potencial que tienen, y presenta un «esqueleto» sobre el que se puede empezar a construir otros sistemas más complejos, que no tienen que ser necesariamente del mismo dominio.

Una mejora que aumentaría drásticamente la calidad de diagnóstico es que funcione de un modo probabilístico. Actualmente, la lógica a la hora de diagnosticar un hecho es completamente binaria: en el caso de las enfermedades, el pez tiene que presentar todos y cada uno de los síntomas, ¿pero qué ocurre cuando de los cuatro síntomas que presenta una enfermedad solamente tiene tres? ¿O si tiene esos tres, y otro síntoma correspondiente a otra enfermedad? Es en casos como estos, donde no se puede determinar una única respuesta con 100% de seguridad, donde encontramos las mayores debilidades del sistema.

Más allá de eso, estos sistemas tienen amplio uso en la realidad. Y si bien posiblemente nunca reemplacen en su totalidad a un profesional humano (especialmente en áreas delicadas, como la medicina), han demostrado ser muy útiles como herramientas de consulta, por ej. para diagnosticar ciertas enfermedades, donde tienen mayor precisión que los médicos.

Referencias

- Enfermedades del goldfish
- Sistemas expertos, EcuRed.
- Sistema experto, Wikipedia.

- Sistemas expertos y sus aplicaciones, Tomás León Quintanar, Universidad Autónoma del Estado de Hidalgo.
- Fuente codigo vistas https://docs.google.com/document/d/1j4Lyde4-VRfAIKpuqAQu1Q6oHTg24P7aKpx30Hhv6f0/edit?usp=sharing

Solucionador de Sudoku

Introducción

El Sudoku es un juego que tiene sus fundamentos en la lógica combinatoria. En su forma tradicional, se presenta en una grilla de 9x9, que a su vez está subdividida en regiones o bloques de 3x3. Al iniciar el juego algunos de los casilleros estarán completados, y a estos valores se los conoce como «números dados» o «pistas».

Las columnas de la grilla se etiquetan normalmente con los números del 1-9, mientras que las filas con las letras A-I. Cada columna, fila y región tiene un total de 9 casilleros, y nos referiremos a cualquiera de estos tres elementos como una «unidad». A su vez, los casilleros que componen una unidad son llamados «pares».

El Sudoku se da por finalizado cuando el jugador logra completar todas las unidades con los dígitos del 1 al 9, de manera tal que en la unidad no encontremos el mismo número repetido. Esto implica que todos los pares deben ser distintos, y que cada dígito debe aparecer como mínimo y como máximo una vez por unidad.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Fig. 1. Un puzzle típico de Sudoku (izq.) y su solución (der.)

Las variantes sobre esta versión tradicional pueden incluir un mayor número de celdas, distintos símbolos (por ej. utilizar letras en vez de números), distintas restricciones (por ej. que los elementos también tengan que ser únicos diagonalmente) o cambiar completamente la lógica del juego (por ej. el Sudoku «mayor que»), entre otras.

Además, se dice que un juego de Sudoku está bien formado cuando tiene solución única.

Solución propuesta

Para implementar la solución utilizaremos el lenguaje de orientación funcional Clojure, que pertenece a la familia LISP y está construido sobre la máquina virtual de Java.

La primera parte del código está orientada a crear la estructura de datos que contendrá la grilla y definir funciones auxiliares que podamos necesitar. En la segunda estará definido el núcleo del programa, que es donde se imponen las restricciones que nos permiten alcanzar la solución. Todo el código puede accederse en texto plano desde aquí.

```
(def filas "ABCDEFGHI")
(def columnas "123456789")
(def digitos "123456789")
(def caracteres-casilla-vacía "0.-")
```

También definimos en las variables `filas`, `columnas` y `dígitos` los caracteres que representan las mismas. Entre otras cosas, todo esto nos servirá luego para crear el tablero de juego. Luego tenemos la variable `caracteres-casilla-vacía`. Cuando leamos la entrada para generar el tablero, las casillas que no tengan ningún número (o sea, que no tengan pista, que deben ser completadas) podrán venir descritas por uno de esos tres caracteres, por lo que los dejaremos definidos aquí.

```
(defn prod-cruz [A, B]
  (for [a A b B] (str a b)))

(def casillas (prod-cruz filas columnas))
```

Mediante `ProdCruz` generamos el producto cruz entre dos colecciones `A` y `B`.

Finalmente, `casillas` tendrá la forma `(A1, A2, ..., A9, B1, ..., I9)`, es decir, es donde quedarán almacenados todos los índices de nuestra grilla.

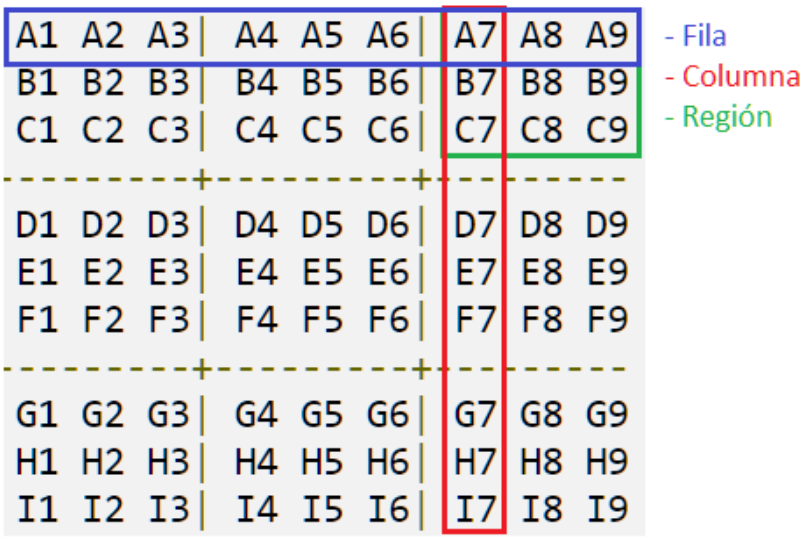


Fig. 2. La estructura utilizada para representar el tablero. Además, se muestra a modo de ejemplo las unidades correspondientes al casillero A7.

```
(def tamaño-region 3)
(def filas-región (partition tamaño-region filas))
(def columnas-región (partition tamaño-region columnas))
```

Estas tres variables nos van a permitir crear las regiones de la grilla. Primero definimos `tamaño-region`, a la cual le asignamos un valor de 3, ya que nuestras regiones tienen dimensiones de 3x3. Luego particionamos las variables `filas` y `columnas` en 3, resultando `filas-región` igual a ((A, B, C), (D, E, F), (G, H, I)) y `columnas-región` ((1, 2, 3), (4, 5, 6), (7, 8, 9)).

Más adelante realizaremos el producto cruz entre estos elementos, y de esa forma obtendremos los índices de las regiones. Por ejemplo, el producto cruz entre el primer elemento de `filas-región` (A, B, C) y el primer elemento de `columnas-región` (1, 2, 3) nos va a dar como resultado (A1, A2, ..., B1, ..., C3), es decir, las casillas de la región que está en la esquina superior izquierda del tablero (ver Fig. 2).

```
(def lista-unidades (map set (concat
                       (for [c columnas] (prod-cruz filas [c]))
                       (for [f filas] (prod-cruz [f] columnas))
                       (for [fs filas-región
                             cs columnas-región] (prod-cruz fs cs)))))
```

La función `lista-unidades` genera un conjunto con todas las unidades de la grilla. Por ejemplo, un elemento de `lista-unidades` es {E1, A1, C1, B1, F1, D1, G1, H1, I1}, que representa la primera columna.

El resultado del primer `for` serán los índices de las casillas pertenecientes a cada columna de la grilla, es decir: (A1, B1, ...) para la primera columna, (A2, B2, ...) para la segunda, etc.

El segundo hace lo mismo para las filas: (A1, A2, ...), (B1, B2, ...), ...

Y el tercero lo mismo para las regiones: (A1, B1, C1, A2, ..., C3), ...

De esta forma, al unir todas estas variaciones lo que estamos obteniendo es la unión de filas, columnas y regiones, es decir, la unión de todas las unidades de la grilla. Luego las unidades son unificadas en una misma colección gracias a la función `set`, que además elimina los elementos repetidos que hayan sido generados.

```
(defn diccionario [x] (apply sorted-map (apply concat x)))
(defn conjunto [x] (apply sorted-set (apply concat x)))
```

Las funciones `diccionario` y `conjunto` nos servirán para crear colecciones ordenadas de estos mismos tipos, a partir de una entrada `x`.

```
(def unidades (diccionario (for [c casillas]
                             [c (for [u lista-unidades :when (u c)] u)])))

(def pares (diccionario (for [c casillas]
                          [c (disj (conjunto (unidades c)) c)])))
```

Las variable `unidades` almacena en la clave de un diccionario (o «mapa») todos los índices de la grilla (A1, A2, ...), y en la parte del valor las unidades de las que forma parte esa casilla. Por ejemplo, cada el caso de A1 va a contener la primera columna, la primera fila y la región superior izquierda:

{A1 (#{E1 A1 C1 B1 F1 D1 G1 H1 I1}, #{A5 A1 A3 A6 A9 A7 A4 A2 A8}, #{A1 A3 C1 B1 C2 B3 C3 A2 B2})

Para esto, por cada casilla `c` recorre todos los elementos de `lista-unidades`, y si `c` forma parte del elemento que está siendo revisado, lo agrega en la clave correspondiente de `unidades`.

Por su parte, `pares` funciona de manera similar: cada clave del diccionario será un índice de la grilla, y en el valor contiene una colección con todas las casillas con las que comparte una unidad. Por ejemplo, la entrada para el casillero A2 será:

A2 #{A1 A3 A4 A5 A6 A7 A8 A9 B1 B2 B3 C1 C2 C3 D2 E2 F2 G2 H2 I2}

Notar que a diferencia de `unidades`, la clave aquí no se repite en el valor debido a la función `disj`, que devuelve la diferencia entre los conjuntos.

Estas dos variables serán fundamentales luego cuando recorramos la grilla para comprobar los valores guardados, asignar uno nuevo en una casilla, etc.

```
(defn filtrar [cadena] (filter (set (concat dígitos caracteres-casilla-vacía))
cadena))

(defn inicializar-grilla [] (diccionario (for [c casillas] [c,(atom dígitos)])))

(defn válidos? [colección] (every? identity colección))

(defn grilla-inicial [cadena]
  (let [cadena (filtrar cadena)
        grilla (crear-grilla)]
```

```
  (if (válidos? (for [[casilla nro]
      (zipmap casillas cadena) :when ((set dígitos) nro)]
      (asignar! valores casilla nro)))
    grilla
    false)))
```

Las funciones `filtrar`, `inicializar-grilla`, `verdareros?` y `grilla` se encargan, a grosso modo, de tomar la grilla con el problema de entrada y prepararla para comenzar a buscar la solución. A continuación describiremos más qué realiza cada función:

1. `filtrar` toma el problema de entrada, que puede venir en distintos formatos de cadena de texto (ver Fig. 3), y remueve todos los caracteres que no formen parte de `dígitos` o `caracteres-casilla-vacía`, limpiando de esta forma la entrada.

```
"4.....8.5.3.........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......"

"""
400000805
030000000
000700000
020000060
000080400
000010000
000603070
500200000
104000000"""

"""
4 . . |. . . |8 . 5
. 3 . |. . . |. . .
. . . |7 . . |. . .
------+------+------
. 2 . |. . . |. 6 .
. . . |. 8 . |4 . .
. . . |. 1 . |. . .
------+------+------
. . . |6 . 3 |. 7 .
5 . . |2 . . |. . .
1 . 4 |. . . |. . .
"""
```

Fig. 3. Posibles formatos de entrada que aceptará el programa.

2. `inicializar-grilla` nos devuelve una grilla limpia para comenzar a resolver el problema. Es una estructura de datos de tipo diccionario, que tendrá en las claves lo que generamos en `casillas` (A1, A2, ..., I9) y en el valor todos los valores de `dígitos`, o sea, 1, 2, ..., 9.

3. `válidos?`, por su parte, nos ayuda a detectar cuando un tablero que es dado como entrada es inválido y no se puede llegar a una solución. Esto puede ocurrir por ej. cuando un valor está repetido en una misma unidad.

4. Y finalmente `grilla` se vale de todas las funciones anteriores para:

    a. Recibir el tablero de entrada y extraer del mismo los valores relevantes (o sea, qué casillas tienen pistas y cuáles están vacías).

    b. Crear la estructura de datos (la «grilla») sobre la cual se ejecutarán los algoritmos e iremos generando la solución.

c. Asignar en esta grilla los valores que fueron suministrados en la entrada, y eliminar los mismos de las unidades correspondientes. Por ejemplo, si una de las pistas ingresadas es que A1 = 1, tengo que eliminar el valor 1 de todas las unidades de las que forma parte esta casilla, quedando en estas (en principio) el resto de `dígitos` (2, 3, ... 9), ya que estos valores vienen cargados de antemano.

A partir de este punto comienza la lógica que resolverá el problema, por lo que conviene introducir la estrategia de asignación que implementará el programa. La misma se basa en dos principios:

1. Si una casilla tiene asignada un único valor, eliminar ese valor de sus pares. Por ejemplo, si sabemos que C1 = 4, entonces podemos eliminar 4 como valor candidato de los pares de C1.

2. Si en una unidad solamente se puede poner un valor en una sola casilla, ponerlo ahí. Continuando con el ejemplo, si sabemos 3 no puede ir en ninguna columna desde C3 a C9, y C1 ya tiene asignada 4, entonces no hay más alternativa que C2 = 3.

Cada vez que se apliquen estos dos principios se produce en todo el tablero un «efecto dominó», ya que al eliminar valores de entre los posibles pueden surgir nuevas asignaciones o eliminaciones, lo que iniciaría el proceso nuevamente.

```
(defn asignar! [grilla casilla nuevo-dígito]
  (if (válidos?
    (for [dígito @(grilla casilla) :when (not (= digito nuevo-dígito))]
      (eliminar! grilla casilla dígito)))
    grilla
    false))
```

Asigna `nuevo-dígito` en `casilla`, y además ejecuta todas las actualizaciones que puedan haberse generado a partir de esa asignación.

Como en un principio las casillas a completar ya vienen cargadas con todos los dígitos, lo que hace en realidad es eliminar de la casilla todos los valores, excepto el que se quiere asignar. En caso de que la asignación genere una grilla inválida, devuelve falso.

```
(defn eliminar! [grilla casilla dígito]
  (if (not ((set @(grilla casilla)) dígito)) grilla     ; (1) Ya está eliminado
      (do
        (swap! (grilla casilla) #(. % replace (str dígito) ""))     ; Remover
        (if (= 0 (count @(grilla casilla)))
          false     ; (2) Grilla inválida
          (if (= 1 (count @(grilla casilla)))     ; (3) Un solo valor posible
```

```
            (let [dígito-aux (first @(grilla casilla))]
              (if (not (verdaderos? (for [casilla-aux (pares casilla)]
               (eliminar! grilla casilla-aux dígito-aux))))
                false
                (comprobar! grilla casilla dígito)))
            (comprobar! grilla casilla dígito))))))
```

Esta función es llamada por `asignar!` para eliminar un valor de una casilla. A la hora de realizar esto existen tres casos:

1. Que el valor ya no esté en la casilla porque fue eliminado antes, en cuyo caso retorna sin hacer nada.
2. Que luego de eliminar el valor la casilla no cuente con ningún valor posible, o sea que se hayan eliminado todos los valores. Esto significa que se alcanzó una grilla inválida, por lo que esta rama de la solución quedará descartada devolviendo `false` al bucle principal.
3. Que luego de eliminar el dígito de la casilla, la misma quede con un único valor posible. En este caso, este valor quedará asignado y se eliminará de los pares de la casilla. Para esto se llama recursivamente a `eliminar!`.

Ahora bien, la estrategia de asignación implementada por estas funciones y explicada anteriormente no alcanza para los sudokus de mayor dificultad. Estos tienen la particularidad de que en cierto punto no vamos a poder seguir filtrando valores, por lo que las casillas pueden tener más de único valor posible (ver Fig. 4). Llegada esta situación, el algoritmo necesita recurrir a otro recurso además de la mera deducción.

```
  4      1679   12679  |  139    2369    269   |  8     1239   5
26789    3    1256789  | 14589   24569   245689 | 12679   1249  124679
 2689   15689  125689  |   7     234569  245689 | 12369  12349  123469
-----------------------+-----------------------+-----------------------
 3789    2     15789   | 3459    34579   4579   | 13579   6     13789
 3679   15679  15679   | 359     8       25679  |  4     12359  12379
36789    4     56789   | 359     1       25679  | 23579  23589  23789
-----------------------+-----------------------+-----------------------
 289    89     289     |  6      459     3      | 1259    7     12489
  5    6789    3       |  2      479     1      |  69     489    4689
  1    6789    4       | 589     579     5789   | 23569  23589  23689
```

Fig. 4. Intento fallido de resolver un sudoku de mayor dificultad.

Una posible solución para «desbloquear» el algoritmo y que pueda seguir avanzando es sencillamente tomar un valor de entre los posibles, asignarlo y tratar desde ahí de llegar a la solución. Más detalladamente:

1. Tomaremos un casillero de los que no están resueltos, es decir los que tienen más de un posible valor. Para que la probabilidad de acertar sean mayores, seleccionaremos primero los que cuentan con menor cantidad de valores, por ej. $G_2 = \{8, 9\}$ en la Fig. 4. Al tener dos posibles valores, uno de ellos debe ser el verdadero. Si seleccionáramos los más grandes, nuestro algoritmo requeriría mayor cantidad de iteraciones para dar con el valor correcto.
2. Seleccionada la casilla (la cual llamaremos «pivote»), le asignamos un número de entre los posibles (en orden numérico, en este caso no es importante cómo los seleccionemos) y verificaremos si a partir de ese estado se puede construir una solución. Si no es posible, ese camino se descarta y retrocedemos para probar con el siguiente valor del pivote.

Esta búsqueda es por naturaleza en profundidad, ya que por cada casilla probamos todos los valores posibles antes de avanzar con la siguiente.

```
(defn copiar-grilla [grilla] (diccionario (for [clave (keys grilla)] [clave (atom
@(grilla clave))])))

(defn solucionar
  ([grilla] (solucionar grilla ""))
  ([grilla, iteración]
   (if grilla
     (if (verdaderos? (for [c casillas] (= 1 (count @(grilla c)))))
       grilla       ; (1)
       (let [pivote ; (2)
             (second (first (sort
                             (for [c casillas :when (> (count @(grilla c)) 1)]
                               [(count @(grilla c)), c])))]
         (let [resultados (for [nro @(grilla pivote)]
                            (do
                              (solucionar (asignar! (copiar-grilla grilla) pivote nro)
(str iteración nro))))]
           (some identity resultados))))
     false)))
```

La función `copiar-grilla` tiene el propósito de generar una copia de la grilla, para preservar su estado actual en caso de que vayamos a buscar una solución, fallemos y tengamos que tomar otra rama.

Finalmente, en `solucionar` llamaremos a `asignar!` para solucionar el problema. Esta función está separada en dos ramas, que se obtienen luego de verificar si la cantidad de valores para todas las casillas es igual a uno.

- Si esto es verdad sencillamente devolvemos la grilla, ya que si todas las casillas tienen exactamente un valor significa que ya está resuelta.

- Si no es verdad, llamaremos recursivamente a `solucionar`, pasándole una copia de la grilla con un pivote mediante `copiar-grilla`. De esta forma podremos desbloquear el estado de la grilla y alcanzar la solución.

Pruebas finales

Y ahora por fin estamos en condiciones de probar el programa. No mencionaremos aquí las funciones utilizadas para dibujar por pantalla la grilla, pero las mismas están presentes en el código fuente.

```
(defn probar-solucion [problema]
     (do
       (println "\n:: Problema:")
       (imprimir-problema problema)
       (println "\n:: Solución:")
       (imprimir-tablero (solucionar (grilla-inicial problema)))))

(probar-solucion sudoku-dificil)
```

```
:: Problema:
850002400
720000009
004000000
000107002
305000900
040000000
000080070
017000000
000036040

:: Solución:
 8   5   9 | 6   1   2 | 4   3   7
 7   2   3 | 8   5   4 | 1   6   9
 1   6   4 | 3   7   9 | 5   2   8
-----------+-----------+-----------
 9   8   6 | 1   4   7 | 3   5   2
 3   7   5 | 2   6   8 | 9   1   4
 2   4   1 | 5   9   3 | 7   8   6
-----------+-----------+-----------
 4   3   2 | 9   8   1 | 6   7   5
 6   1   7 | 4   2   5 | 8   9   3
 5   9   8 | 7   3   6 | 2   4   1
```

Fig. 5. Resolución de un sudoku difícil.

Claramente no alcanza con probarlo con un único problema, por eso hemos descargado un archivo de texto con 95 sudokus de dificultad difícil para suministrárselo al programa como entrada. El resultado de las pruebas fue exitoso, y puede consultarse aquí.

Referencias

- Solving Every Sudoku Puzzle, Peter Norvig.
- Learning Clojure: Sudoku Solver, John Lawrence Aspden.

Informe de Horóscopo

Introducción

En este informe se presenta el desarrollo de un sistema que permite al usuario conocer su signo zodiacal basándose en su fecha de nacimiento. Si bien esta funcionalidad no es nada del otro mundo, creemos que lo importante del proyecto está en que además de una implementación en Prolog, ofreceremos una alternativa en una librería llamada Drools.

Drools, es un sistema de gestión de reglas construido sobre Java, muy similar a Prolog. Su funcionamiento está basado en el encadenamiento hacia delante (parte de las reglas y trata de llegar al objetivo) y el encadenamiento hacia atrás (parte de la conclusión y trata de llegar hasta las reglas). El corazón de Drools es el algoritmo Rete, muy utilizado en el mundo de los sistemas expertos, ya que permite aplicar eficientemente una gran cantidad de reglas sobre muchos objetos.

Estas características hacen que sea muy similar a Prolog, ya que ambos basan su ejecución en la unificación y el backtracking («vuelta atrás»), lo que les permite hacer una búsqueda sistemática a través de todas las configuraciones posibles dentro de un espacio de búsqueda. Para lograr esto, los algoritmos de backtracking construyen un árbol donde contemplan todas las posibles soluciones candidatas.

De esta manera, ofreceremos una implementación que utiliza Prolog para la lógica y Java para las interfaces, y mostraremos también —de manera más resumida— una posible implementación en Drools, al menos para conocer un poco esta librería.

Solución propuesta (Prolog)

El punto de entrada del programa es la siguiente interfaz, la cual está conformada por un campo para introducir la fecha de nacimiento de la persona. Luego se hace clic en el botón «Ver», y se mostrará el resultado de la consulta, es decir su signo.

El código de la interfaz decidimos no agregarlo para concentrarnos en la lógica de la solución del problema, de manera que no quede tan extenso y sea más concreto, pudiendo detallar más en profundidad lo que nos interesa.

Lo primero que haremos es definir los hechos para cada signo zodiacal en el predicado `horoscopo`, y en cada uno de sus cláusulas definiremos un signo del zodíaco.

```
horoscopo(aries,21,3,20,4).
horoscopo(tauro,21,4,21,5).
horoscopo(geminis,22,5,21,6).
horoscopo(cancer,22,6,22,7).
horoscopo(leo,23,7,22,8).
horoscopo(virgo,23,8,22,9).
horoscopo(libra,23,9,22,10).
horoscopo(escorpio,23,10,21,11).
horoscopo(sagitario,22,11,21,12).
horoscopo(capricornio,22,12,20,1).
horoscopo(acuario,21,2,20,3).
```

Como vemos, las cláusulas siguen el siguiente formato:

`horóscopo(nombre signo, día inicio, mes inicio, día fin, mes fin).`

Donde:
- **nombre signo**: son todos los signos zodiacales;
- **día inicio** y **mes inicio**: son el día y mes de inicio del calendario que representa un signo;
- **día fin** y **mes fin**: son el día y mes de fin del calendario que representa un signo.

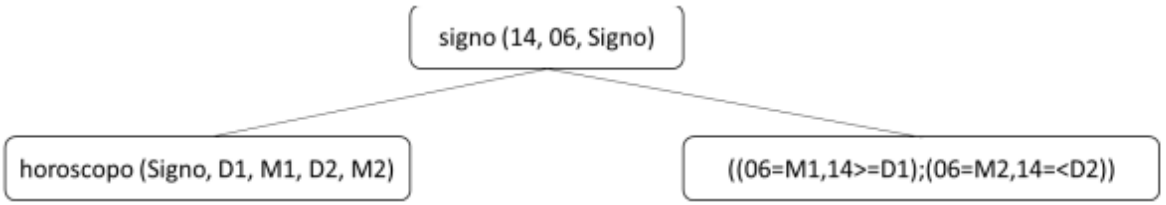
Luego de crear los hechos definiremos la regla `signo`, que determina el signo zodiacal del usuario. Esta regla será el núcleo de nuestro motor de inferencia:

```
signo(Dia,Mes,Signo) :-
    horoscopo(Signo,D1,M1,D2,M2),
    ((Mes=M1, Dia>D1); (Mes=M2,Dia=<D2)).
```

Partimos de la hipótesis de que el signo de un día y mes sea verdadero si se cumple algún hecho definido de `horóscopo`, que el mes ingresado en la hipótesis sea igual al mes inicial del hecho, y el día ingresado mayor o igual al día inicial del hecho; o sino, que el mes ingresado sea igual al mes final del hecho y que el día ingresado sea menor o igual día final del hecho.

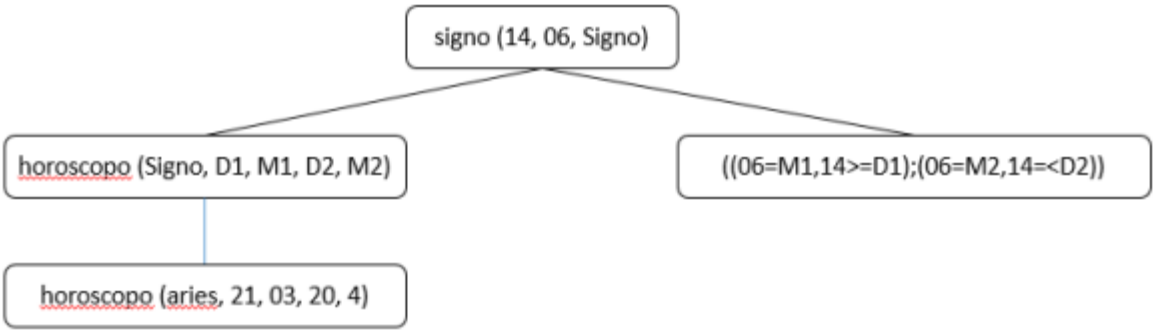
Para entender mejor este proceso, mostraremos el siguiente ejemplo acompañado del árbol de decisión que se crea a partir de la consulta, la cual es: `signo(14, 06, Signo)` (14 de Julio). Para que esto sea verdad, se deben encontrar los valores de `Signo` que cumplan las

restricciones impuestas, es decir, Prolog tomará la regla `signo` y buscará en su base de conocimientos los signos que cumplan la condición de mes 6 y dia 14.



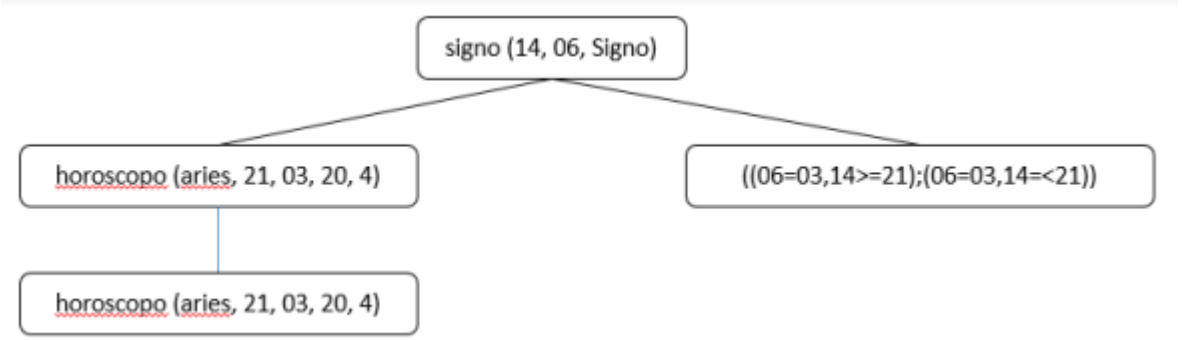
Para que esa hipótesis sea verdadera se deben cumplir las dos ramas del árbol, que representan el cuerpo del predicado `signo`.

Primero se consulta en la base de conocimiento por el hecho `horoscopo(Signo, D1, M1, D2, M2)`, empezando desde el primer hecho declarado:



Así obtiene la primera respuesta, que es: `Signo=Aries, D1=21, M1=3, D2=20, M2=4`.

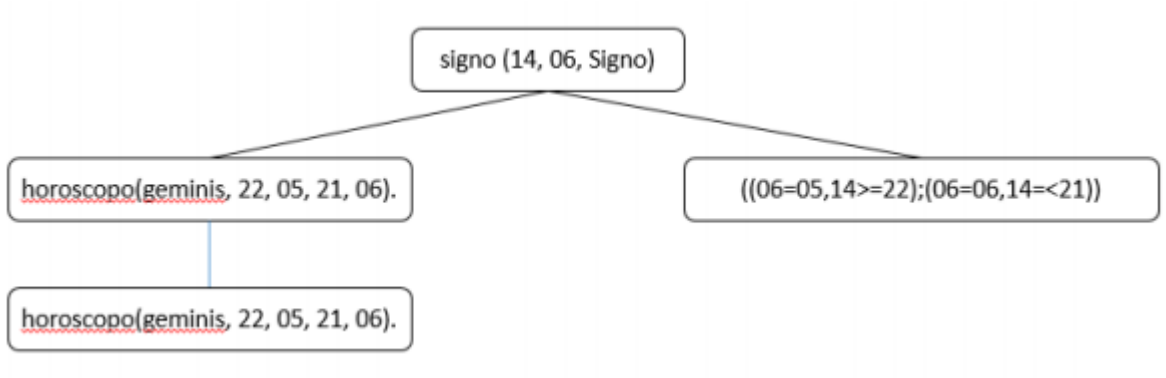
De esta forma obtenemos de la primera hipótesis una serie de valores que nos van a servir para cargar la segunda parte del predicado:



Entonces llegamos a obtener valores verdaderos para la primera rama del árbol, pero no para la segunda, ya que al compararlos con los valores ingresados por el usuario (mes 6, día 14) vemos que pueden compararse exitosamente.

Esto lleva a que se produzca un fallo, por lo que se replanteará la consulta (mediante backtracking) hasta que se encuentre una solución exitosa o hasta quedarnos sin opciones de búsqueda. En este caso, se continúa con el siguiente hecho en el orden definido.

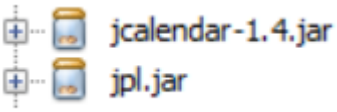
Eventualmente llegaremos al hecho `horoscopo(geminis,22,05,21,06)`, esto es: `Signo=geminis, D1=22, M1=5, D2=21, M2=6`. Al cargar estos valores en la segunda rama obtenemos el siguiente árbol:



Y como puede verse, en este caso se encuentra un éxito en la segunda rama, entonces, siendo la primera y segunda rama verdadera, se tiene un signo el cual satisface la hipótesis: Géminis. Aquí se realiza backtracking hasta finalizar con los hechos para comprobar si no existe otro caso de éxito. Como vimos, en Prolog se utiliza la inferencia de encadenamiento hacia atrás, este parte desde una hipótesis creando ramas con las distintas reglas que se tengan hasta llegar a los hechos que cumplan con la hipótesis y comprobar que no existen más hechos que cumplan la hipótesis.

Fig. 1. Respuesta obtenida al consultar por el 4 de junio.

Para la implementación en la aplicación Java se usaron las librerías:



`jcalendar` sencillamente implementa el control que le permite al usuario seleccionar la fecha. `jpl`, por su lado, nos permite conectar de forma bidireccional a Java y a Prolog,

dándonos por un lado acceso al sinnúmero de librerías y código reutilizable de Java, y la solidez y seguridad de Prolog por el otro.

```
private void ver() {
    Calendario fecha = new Calendario(jDCfechaN.getCalendar());
    String consulta = "consult('horoscopo.pl')";
    Query qConsulta = new Query(consulta);
    System.out.println(qConsulta.hasSolution());
    String consultaSigno = "signo(" + fecha.getDia() + "," + fecha.getMes() + ",Signo)";
    Query qConsultaSigno = new Query(consultaSigno);
    System.out.println(qConsultaSigno.toString());
    signo = "" + qConsultaSigno.oneSolution().get("Signo");
    System.out.println(signo);
    jLsigno.setText("Su signo del zodiaco es: " + signo);
```

Fig. 2. Enviando la consulta a Prolog desde Java.

En el recuadro rojo (Figura 2) podemos ver cómo se lleva a cabo esto. En la variable `qConsulta` (que es de tipo `Query`, el cual está implementado por `jpl`) «cargamos» el archivo `horoscopo.pl`. Luego en `consultaSigno` generamos la consulta en sí, pasándole como parámetro el día y la fecha seleccionadas por el usuario en el calendario, para luego mostrar por pantalla una respuesta acorde al signo obtenido.

```
run:
true
signo( 16, 2, Signo )
acuario
BUILD SUCCESSFUL (total time: 9 seconds)
```

Fig. 3. Debugging de una sesión de prueba (día 16, mes 2).

Solución propuesta (Drools)

Presentaremos ahora una implementación alternativa, reemplazando a Prolog por Drools para la parte «lógica» de la solución.

Al igual que Prolog, Drools también debe contar para su funcionamiento con una base de conocimientos y un motor de inferencia, el cual es creado mediante reglas de producción o sencillamente *rules*. Estas reglas son piezas de conocimiento y son de la forma: "When (Cuando) se producen algunas condiciones, then (entonces) hacer algunas tareas."

```
when
    Unos hechos sean ciertos
then
    Llega a estas conclusiones
```

Fig. 4. Formato de las reglas en Drools.

Estas reglas parten de unos hechos y según la validez de los mismos llegan a unas conclusiones. Ahora para nuestro sistema experto que determina el signo zodiacal debemos

definir varias cosas. Primero creamos una clase `Persona` en la cual se definen el mes y día de nacimiento:

```
public static class Persona {

    private int dia;
    private int mes;

    public int getDia() {
        return dia;
    }

    public void setDia(int dia) {
        this.dia = dia;
    }

    public int getMes() {
        return mes;
    }

    public void setMes(int mes) {
        this.mes = mes;
    }

}
```

Definida esta clase, crearemos las reglas del sistema, que tendrán la siguiente estructura:

```
rule "signo"
    when
        fecha de nacimiento ingresada cumple la condicion
    then
        mostrar signo zodiacal correspondiente
end
```

De esta forma, definiremos por ej. la regla para el signo aries:

```
rule "aries"
    when
      p : Persona( (dia >= 21 && mes == 3) || ( dia <= 21 && mes == 4))
    then
      JoptionPane.showMessageDialog(null, "Su signo zodiacal es: Aries.",
                            "Horóscopo Software",JOptionPane.INFORMATION_MESAGE);
end
```

Aquí estamos diciendo que la regla con nombre `Aries` se activará cuando el día ingresado sea mayor o igual a 21 y el mes ingresado sea igual a 3; o, cuando el día ingresado sea menor o igual a 20 y el mes ingresado sea igual a 4. Si se cumplen alguna de esas condiciones, entonces se muestra un mensaje diciendo que su signo zodiacal es Aries. Y por último se coloca la palabra clave `end` para indicar el final de la regla.

La definición del resto de las reglas es similar, solamente debemos modificar su nombre, los días que le corresponden y el mensaje a mostrar:

```
rule "capricornio"
    when
        p : Persona( (dia >= 22 && mes == 12) || (dia <= 20 && mes == 1))
    then
        JOptionPane.showMessageDialog(null, "Su signo zodiacal es: Capricornio.",
        "Horoscopo Software ®", JOptionPane.INFORMATION_MESSAGE);
end
```

Conclusiones

Si bien en la implementación que utiliza Drools no encontramos muchos beneficios sobre la de Prolog, queremos de todas formas hacer una comparación entre estas dos herramientas ya que ocupan un campo bastante similar. La diferencia principal que encontramos es que Drools es que es una herramienta de gestión de reglas de negocio, lo que en términos prácticos significa que es una herramienta «más empresarial», a diferencia de Prolog donde tal vez su campo fuerte sea el académico. El ser más empresarial conlleva a que trabaje en un más alto nivel, permitiendo por ej.:

- Crear modelos de datos mediante interfaces gráficas.
- Acceder a visualizaciones que permiten ver cómo las reglas se afectan entre sí.
- Ofrecer mayor integración con otras herramientas empresariales.
- Ofrecer mayor facilidad para almacenar cambios en la base de conocimientos.

Otra característica de Drools es que a la hora de formular las reglas está más cercano a la programación imperativa y sus estructuras de decisión. Al observar las reglas, vemos que siguen el mismo formato que un IF/THEN clásico, lo que puede resultar más amigable para los programadores acostumbrados a este paradigma. Esto proviene en parte de otra de las características fuertes de Drools, no presente en Prolog: el encadenamiento hacia adelante. El encadenamiento hacia adelante básicamente genera todas las inferencias que puede a partir de las condiciones iniciales que se le suministren. A su vez, estas inferencias pueden generar eventos que activen ciertas acciones, por ej. iniciar cierto proceso.

Por su parte, el fuerte de Prolog son las aplicaciones «interrogativas», es decir, se le suministra una consulta al programa con cierto resultado y se le pregunta cuáles son las condiciones que aseguran ese final. Esto es posible gracias a los dos pilares de Prolog, el encadenamiento hacia atrás y la unificación (que también están presentes, aunque de manera más reciente, en Drools).

Referencias

- Proyecto Horóscopo desarrollado en Prolog y Java. https://drive.google.com/file/d/0B1b0YM_QXyyCczR2Um1ZZWhwNFE/view?usp=sharing
- Proyecto Horóscopo desarrollado en Drools y Java. https://drive.google.com/file/d/0B1b0YM_QXyyCalVwOEV3NGV0TWM/view?usp=sharing
- Árbol de búsqueda realizado por Prolog para resolver una hipótesis. https://drive.google.com/file/d/0B1b0YM_QXyyCNU94WTYyMEJOMmc/view?usp=sharing