Trabajo Práctico 2
Programación Lógica

Paradigmas de Lenguajes de Programación — 2do cuat. 2014

Fecha de entrega: 4 de noviembre

1. Introducción

En este TP trabajaremos con *autómatas finitos*. Un autómata es una máquina de estados —en este caso, finitos— cuyo objetivo es reconocer palabras de un lenguaje. Así, un lenguaje se entiende como el conjunto de todas las palabras que reconoce el autómata. En adelante profundizaremos en la descripción de los autómatas en sí y el análisis de las palabras que reconocen.

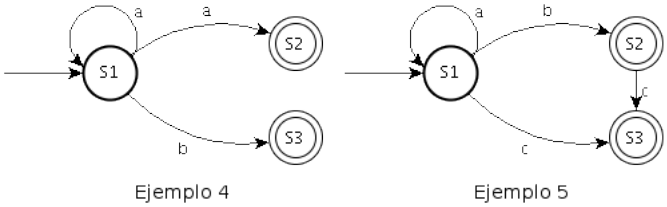

Figura 1: Ejemplos de autómatas finitos

Un autómata consiste en un conjunto de estados, representados en la figura por círculos, y las transiciones entre ellos, representadas por flechas. Cada transición tiene una etiqueta correspondiente a un carácter del alfabeto que se utiliza para ese autómata. Para reconocer una palabra el autómata parte de su estado inicial, el que se indica con una flecha proveniente del exterior, y en cada transición acumula el carácter indicado por la etiqueta correspondiente. La palabra es reconocida si se logra acumular sucesivamente todos sus caracteres en el orden correcto, terminando en un estado final, señalado en la figura con un círculo doble. Puede haber varios estados finales, y se permite pasar por un estado final y seguir acumulando caracteres, pero el final de la palabra debe coincidir con la llegada a un estado final.

Los autómatas pueden ser o no ser determinísticos. Un autómata es determinístico si desde cada estado sale, a lo sumo, una transición con una misma etiqueta. En la figura 1 se aprecia que el autómata 'Ejemplo 4' no es determinístico, ya que hay más de una transición para la etiqueta `a` desde `S1`, y el 'Ejemplo 5' sí lo es. Adicionalmente, ambos pueden reconocer una cantidad infinita de palabras, lo que es resultado de que los autómatas poseen ciclos.

Representación utilizada

Un autómata estará representado con la estructura: `a(inicial,finales,transiciones)`, donde `inicial` es un átomo que indica el estado inicial, `finales` es la lista de estados finales,

y `transiciones` es la lista de transiciones entre estados. A su vez, cada transición es una tupla de la forma `(estado_origen, etiqueta, estado_destino)`.

2. Ejercicios

Se pide diseñar e implementar en Prolog los siguientes predicados, indicando en cada caso si utilizan o no la técnica de *generate and test*.

1. `esDeterministico(+Automata)` que determina si es cierto que para todo estado hay, a lo sumo, una única transición por cada etiqueta.

2. `estados(+Automata,?Estados)` que tiene éxito si en `Estados` se encuentra la lista de los distintos estados del autómata. Si `Estados` está sin instanciar la lista debe generarse en orden creciente y sin repetidos. Por otro lado, si está instanciada el predicado debe tener éxito si es correcta sin importar el orden o que tenga repetidos. Por ejemplo:

```
?- ejemplo(5, A), estados(A, E).
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]),
E = [s1, s2, s3] .

?- ejemplo(5, A), estados(A, [s2,s1,s2,s3]).
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]) .

?- ejemplo(5, A), estados(A, [s2,s1]).
false.
```

3. `esCamino(+Automata, ?S1, ?S2, +Camino)` que determina si `Camino` es o no un camino posible entre `S1` y `S2`. Por ejemplo:

```
?- ejemplo(5, A), esCamino(A, s1, s2, [s1,s2]).
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]) .

?- ejemplo(5, A), esCamino(A, I, F, [s1,s1,s2,s3]).
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]),
I = s1,
F = s3 ;

?- ejemplo(5, A), esCamino(A, s1, s1, [s1]).
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]) .

?- ejemplo(5, A), esCamino(A, s1, s1, [s1,s1,s1,s1,s1,s1]).
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]) .
```

4. Indicar si el predicado anterior es o no reversible con respecto a `Camino` y por qué (en particular en caso de que no lo fuera). Responder en forma de comentario.

5. `caminoDeLongitud(+Automata, +N, -Camino, -Etiquetas, ?S1, ?S2)` que instancia en `Camino` a un camino de longitud `N` dentro de `Automata`, que parte de `S1` hasta `S2` y utiliza las transiciones en `Etiquetas`. Ejemplo:

```
?- ejemplo(5, A), caminoDeLongitud(A, 3, Camino, Etiquetas, Origen, Destino).
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]),
Camino = [s1, s1, s1],
Etiquetas = [a, a],
Origen = Destino, Destino = s1 ;
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]),
Camino = [s1, s1, s2],
Etiquetas = [a, b],
Origen = s1,
Destino = s2 ;
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]),
Camino = [s1, s1, s3],
Etiquetas = [a, c],
Origen = s1,
Destino = s3 ;
A = a(s1, [s2, s3], [ (s1, a, s1), (s1, b, s2), (s1, c, s3), (s2, c, s3)]),
Camino = [s1, s2, s3],
Etiquetas = [b, c],
Origen = s1,
Destino = s3.
```

6. `alcanzable(+Automata, +Estado)` que indica si `Estado` es alcanzable desde el estado inicial. Un estado no se considera alcanzable desde sí mismo a menos que forme parte de un ciclo.

7. `automataValido(+Automata)` que determina si `Automata` cumple los siguientes requisitos:

   a) todos los estados tienen transiciones salientes exceptuando los finales, que pueden o no tenerlas.

   b) todos los estados son alcanzables desde el estado inicial.

   c) el autómata tiene al menos un estado final.

   d) no hay estados finales repetidos.

   e) no hay transiciones repetidas.

   Sugerencia: explorar el meta-predicado `forall/2` para simplificar el código.

Nota: de aquí en más se puede asumir que los autómatas son válidos de acuerdo a esta definición.

8. `hayCiclo(+Automata)` que detecta la presencia de ciclos en un autómata. En caso de haber un ciclo, el programa debe terminar sin devolver más de un resultado. Es decir, no debe darse la posibilidad de presionar `;` para buscar más resultados. A partir de este ejercicio recomendamos distinguir el caso de autómatas con y sin ciclos.

9. `reconoce(+Automata,?Palabra)` que instancia en `Palabra` la lista de etiquetas que se consumen para llegar de un estado inicial a un estado final, o verifica que la palabra indicada sea reconocible por el autómata. `Palabra` podría estar semi-instanciada (por ejemplo, `[p, X, r, X, d, i, _, m, X, s]`), y en este caso deberán instanciarse las variables correspondientes.

10. `palabraMásCorta(+Automata,?Palabra)` que instancia o verifica que `Palabra` sea la más corta que pueda generarse utilizando el autómata. En caso de haber más de una palabra con la misma longitud, se debe ir instanciando de a una por vez.

3. Condiciones de aprobación

El principal objetivo de este trabajo es evaluar el correcto uso del lenguaje PROLOG de forma declarativa para resolver el problema planteado. El código debe estar *adecuadamente* comentado (es decir, los comentarios que escriban deben facilitar la lectura de los predicados, y no ser una traducción al castellano del código). También se debe explicitar cuáles de los argumentos de los predicados auxiliares deben estar instanciados usando `+`, `-` y `?`.

La entrega debe incluir casos de prueba en los que se ejecute al menos una vez cada predicado definido.

En el caso de los predicados que utilicen la técnica de *generate and test*, deberán indicarlo en los comentarios.

4. Algunas consideraciones sobre *generate and test*

Al generar y testear se toma un candidato y se verifica si sirve o no como solución. Para eso hay que decidir cuál es el universo de posibles soluciones. Una mala decisión de este universo puede impactar muy fuertemente en la eficiencia.

Por ejemplo, si se quieren calcular los factores primos de un número N, una forma es probar número por número, en este caso el universo de posibles soluciones son los naturales. Otra forma sería tomar como universo de soluciones posibles los números primos entre 2 y N. En ambos casos el algoritmo de verificación es el mismo: tomar un candidato y ver si divide o no al número en cuestión.

Si el universo de soluciones posibles son todos los naturales, se tardará mucho en encontrar las soluciones y además, al no estar acotado el conjunto de partida, no se terminará nunca. Si en cambio contamos con una forma eficiente de enumerar los números primos y probar con estos, entonces encontraremos las soluciones de manera mucho más rápida.

En muchos otros casos se pueden generar las soluciones de manera directa, sin necesidad de testear posteriormente. Esto es deseable siempre que la generación sea un procedimiento sencillo y relativamente eficiente.

Además de la elección del universo de candidatos es importante tener en cuenta que, por como funciona el algoritmo detrás de PROLOG, **cuanto antes** podemos descartar una solución tanto mejor. Es decir que si al ir construyendo un candidato a solución descubrimos por el camino que no va a servir, podemos evitarnos una o varias ramas de ejecución potencialmente muy largas.

En algunos problemas se puede partir el proceso de generación y testeo en dos o más etapas que se aplican intercaladas. Por ejemplo:

```
esSolucion(X) :- generarParteUno(X1), testearParteUno(X1),
                 generarParteDos(X1,X), testearParteDos(X).
```

Donde `generarParteDos(+X1,-X2)` parte de una solución parcial ya testeada `X1` y la aumenta para conseguir un candidato a solución `X2` que a su vez deberá ser testeado, ya asumiendo que una parte es correcta.

5. Pautas de Entrega

Se debe entregar el código impreso con la implementación de los predicados pedidos. Cada predicado debe contar con un comentario donde se explique su funcionamiento. Cada predicado asociado a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Prolog a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PL]` seguido inmediatamente del nombre del grupo.
- El código Prolog debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto con nombre `tp2.pl`.

El código debe poder ser ejecutado en SWI-Prolog. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado. Los objetivos a evaluar en la implementación de los predicados son:

- corrección,
- declaratividad,
- reutilización de predicados previamente definidos
- Uso de unificación, backtracking, generate and test y reversibilidad de los predicados que correspondan.
- **Importante**: salvo donde se indique lo contrario, los predicados no deben instanciar soluciones repetidas. Vale aclarar que no es necesario filtrar las soluciones repetidas si la repetición proviene de las características de la entrada.

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

6. Referencias y sugerencias

Como referencia se recomienda la bibliografía incluída en el sitio de la materia (ver sección *Bibliografía* → *Programación Lógica*).

Se recomienda que, siempre que sea posible, utilicen los predicados ISO y los de SWI-Prolog ya disponibles. Recomendamos especialmente examinar los predicados y metapredicados que figuran en la sección *Otros* de la página de la materia. Pueden hallar la descripción de los mismos en la ayuda de `SWI-Prolog` (a la que acceden con el predicado `help`). También se puede acceder a la documentación online de SWI-Prolog.