Fecha 14 de Abril de 2016

Práctica 1: Programación Lógica Pura

MEMORIA DE LA PRÁCTICA
ETSIINF UPM

- Fernández Castro, Roberto Daniel - 110069
- Alobuela Collaguazo, Maria Jose - 130227
- Martinez Alvaro, Gerald Renzo - 090199

Índice de Títulos

Introducción

En esta memoria se explicará la solución que proponemos a la primera práctica de la asignatura, en la que se emplea programación lógica pura para su realización. Dicha práctica está dividida en dos partes.
Por un lado, tenemos la primera parte en la que se pide programar el predicado llamado cells/2, que tiene una lista como parámetro de entrada y otra lista con el parámetro de salida, que es el resultado de aplicar las reglas del autómata celular planteado.
Por otro lado, en la segunda parte se pide programar los predicados arbolBalanceadoPar/1 y arbolAmplificado/2, en el primer predicado se comprueba que la suma de los pesos pares de su árbol izquierdo es igual a la suma de los pesos pares de su árbol derecho y en el segundo predicado se incrementa el peso de los nodos en el número de veces que indica la raíz del árbol origen. Estos dos predicados son las dos etapas por las que pasa el algoritmo de Codificación Huffman, el cual es uno de los más utilizados para la comprensión de datos.

En el documento se empezará mostrando y explicando los predicados de apoyo que han sido utilizados para la realización de las correspondientes partes de la práctica. A continuación se mostrará tanto para la primera como para la segunda parte de la práctica el código con las correspondientes explicaciones, luego se mostrarán las consultas que hemos realizado para comprobar el correcto funcionamiento del código y por último unas conclusiones finales.

Implementaciones Básicas

Predicados de apoyo que nos han servido para realizar pequeñas implementaciones en el código de la práctica.

Utilizados en Autómata Celular

Predicado my_append/3

El siguiente predicado concatena dos listas, dejando el resultado en una nueva lista.
Los argumentos del predicado son:

- [X|Y]: argumento de tipo lista.
- Z: argumento de tipo lista.
- [X|R]: argumento de tipo lista.

```
my_append([ ],X,X).
my_append([X|Y],Z,[X|R]):- my_append(Y,Z,R).
```

Utilizados en Codificación Huffman Plog

Predicado nat/1

Verifica que los números son naturales.
Los argumentos del predicado son:

- s(X): Número de Peano.

```
nat(0).
nat(s(X)):-nat(X).
```

Predicado plus/3

Suma los elementos X e Y, dejando el resultado en Z.
Los argumentos del predicado son:

- s(X): Número de Peano.
- Y: Número de Peano.
- s(Z): Número de Peano.

```
plus(0,X,X):-nat(X).
plus(s(X),Y,s(Z)):- plus(X,Y,Z).
```

Predicado less/3

Comprueba que X es menor que Y.
Los argumentos del predicado son:

- s(X): Número de Peano.
- s(Y): Número de Peano.

```
less(0,s(X)):- nat(X).
less(s(X),s(Y)) :- less(X,Y).
```

Predicado mod/3

Realiza el módulo de X e Y, dejando el resultado en Z.
Los argumentos del predicado son:

- X: Número de Peano.
- Y: Número de Peano.
- Z: Número de Peano.

```
mod(X,Y,X) :- less(X,Y).
mod(X,Y,Z) :- plus(X1,Y,X), mod(X1,Y,Z).
```

Predicado times/3

Multiplica los elementos X e Y, dejando el resultado en Z.
Los argumentos del predicado son:

- X: Número de Peano.
- s(Y): Número de Peano.
- Z: Número de Peano.

```
times(X,0,0):-nat(X).
times(X,s(Y),Z):- times(X,Y,Acc), plus(Acc,X,Z).
```

Parte Uno: Autómata celular

Reglas: Fichero automaton.pl

Estas son las reglas que estarán en nuestra base de hechos, para ello utilizamos un predicado regla/4 como se especifica en el enunciado de la práctica.

```
regla(o,o,o,o).
regla(x,o,o,x).
regla(o,x,o,o).
regla(o,o,x,x).
regla(x,o,x,x).
regla(x,x,o,x).
regla(o,x,x,x).
regla(x,x,x,o).
```

Programa Principal

Predicado add_zeros/2

Predicado que añade el elemento “o” al principio y al final de la lista pasada como argumento, para ello empleamos my_append, que nos devolverá en el argumento Res una nueva lista con los elementos ya añadidos.
Los argumentos del predicado son:

- L: argumento de tipo lista.
- R: argumento de tipo lista.

```
add_zeros(L,Res):- my_append([o],L,L2), my_append(L2,[o],Res).
```

Predicados check_head/1 y check_tail/1

Los predicados verifican que los elementos de una lista empiecen por “o,x” y terminen por “x,o”.
Los argumentos de ambos predicados son:

- L: argumento de tipo lista.

```
check_head([o,x|L]):- check_tail(L).
check_tail([x,o|[]]).
check_tail([_|L]):- check_tail(L).
```

Predicado Iterador/3

El siguiente predicado recorre una lista (parámetro de entrada), sustituyendo cada elemento por su correspondiente célula, según la regla del autómata, y almacenándolo en una nueva lista(parámetro de salida). Para ello llamaremos a regla, que comprobará a qué célula cambia, y con my_append lo concatenamos en una nueva lista, pasando esta lista recursivamente a iterador.
Los argumentos del predicado son:

- [X,Y,Z|H]: Argumento de tipo lista. Es la lista a iterar.
- Acc: Argumento de tipo lista donde se va acumulando la nueva lista según las reglas.
- L: Argumento de tipo lista. Es la lista final.

```
iterador([_,_],Acc,Acc).
iterador([X,Y,Z|H],Acc,L):-
      regla(X,Y,Z,Res),
      my_append(Acc,[Res],NewAcc),
      iterador([Y,Z|H],NewAcc,L).
```

Predicado cells/2

Dada una lista (parámetro de entrada), se obtiene otra (parámetro de salida) cuyos elementos son el resultado de aplicar las reglas del autómata.

El algoritmo empleado para la realización del predicado cells/2 consisten en:
Primero verificamos una de las condiciones que se piden, que es que la lista empiece por ”o,x” y termine por “x,o”, para ello utilizamos el predicado check_head/1, cuyo argumento de entrada será L1, una vez hecho eso, añadimos a L1 el elemento “o” tanto al principio como al final ya que así la lista se hará finita (pues suponemos que los elementos a la derecha e izquierda son “o”), esto lo hacemos con add_zeros/2. La recursividad se encuentra en el predicado iterador/3, pues es el que va a recorrer la lista. Este predicado coge los tres primeros elementos de la lista para ser sustituidos por un elemento, que será el resultado de aplicar una determinada regla del autómata, dicho elemento se almacena en una nueva lista. Después de recorrer la lista, se vuelve a cells con una lista (Res2) que contendrá los nuevos elementos. Por último se vuelve a añadir el elemento “o” al principio y final de la lista (L2).
Los argumentos del predicado son:

- L1: argumento de tipo lista.

- L2: argumento de tipo lista.

```
cells(L1,L2):-
      check_head(L1),
      add_zeros(L1,Res1),
      iterador(Res1,[],Res2),
      add_zeros(Res2,L2).
```

```
check_head(L1) ^ add_zeros(L1,Res1) ^ iterador(Res1,[],Res2) ^
add_zeros(Res2,L2) → cells(L1,L2)
```

Consultas Realizadas para cells/2

En las siguientes pruebas, el argumento de entrada se corresponde con una lista que contiene determinados elementos, el resultado de llamar al predicado cells, será obtener una nueva lista (argumento de salida) en la que aparezcan nuevos elementos, que son el resultado de aplicar las reglas del autómata a los elementos originales.

```
?- cells([o,x,x,x,o,o,o,x,o],R).

R = [o,x,x,o,x,x,o,x,o,x,o] ?;

no

?- cells([o,x,x,o,x,x,o,x,o],R).

R = [o,x,x,x,x,x,x,x,o,x,o] ?;

no
```

En las siguientes pruebas verificamos que no se admite una lista que no empiece por “o,x” y termine por “x,o”.

```
?- cells([o],R).

no.

?- cells([o,o,x,o,x,o,o],R).

no.
```

En la siguiente prueba, dada una lista resultado, queremos obtener la lista original, aparentemente lo hace, pero ciao se queda parado.
?- cells(R,[o,x,x,o,x,x,o,x,o,x,o]).

R = [o,x,x,x,o,o,o,x,o] ? ;

R = [o,x,x,x,o,x,o,x,o] ?

Comprobando en el debug, hayamos el fallo. Resulta que al comprobar my_append, ocurre lo siguiente:

```
38  6  Fail: user:my_append([x,x],[o],[o,x,x,o,x,o,x,o]) ?
37  5  Fail: user:my_append([x,x,x],[o],[x,o,x,x,o,x,o,x,o]) ?
36  4  Fail: user:my_append([x,x,x,x],[o],[x,x,o,x,x,o,x,o,x|...]) ?
```

Al pasarle la lista final y concatenar, no coinciden los resultados, por lo que falla. El programa da la solución correcta porque realiza un Redo hasta dar con ella. No supimos dar con la solución para que lo hiciese correctamente, por lo que este es el único fallo de nuestra práctica.

Parte Dos: Codificación Huffman Plog

Primera Etapa del Algoritmo

Predicado arbolBalanceadoPar/1

El siguiente predicado tiene como argumento de entrada un árbol, en el que se comprueba que la suma de los pesos pares de su árbol izquierdo es igual a la suma de los pesos pares de su árbol derecho. Para ello utilizamos otro predicado rama/4 al que se le pasa por un lado el hijo izquierdo y por otro el hijo derecho. Le pasamos a rama un acumulador = 0 para inicializarlo. En Res obtendremos la suma de ambas ramas y verificamos si son iguales.

```
arbolBalanceadoPar(void).
arbolBalanceadoPar(tree(par(_,_),Left,Right)):-
      rama(Left,0,0,Res),
      rama(Right,0,0,Res).
```

```
rama(Left,0,0,Res) ^ rama(Right,0,0,Res) →
arbolBalanceadoPar(tree(par(_,_),Left,Right))
```

Primer Predicado rama/4

Realiza la suma de los pesos pares del árbol, su parámetro de entrada es el hijo izquierdo, los acumuladores Acc1,Acc2 y el parámetro de salida es Res. En este predicado primero se hace una comprobación de si el peso del nodo es par, para ello utilizamos mod/3, a continuación en Acc1 se acumula los pesos del hijo izquierdo y en Acc2 se acumula los pesos del hijo derecho, posteriormente procedemos a sumar hijo izquierdo con el hijo derecho y finalmente se suma el nodo.

```
rama(void,_,_,_).
rama(tree(par(_,N),Left,Right),Acc1,Acc2,Res):-
      mod(N,s(s(0)),0),
      rama(Left,Acc1,Acc2,Acc1),
      rama(Right,Acc1,Acc2,Acc2),
      plus(Acc1,Acc2,N1),
```

```
        plus(N1,N,Res).
```

Segundo Predicado rama/4

Realiza la suma de los pesos impares del árbol. Los parámetros de entrada son el hijo derecho, Acc1, Acc2, recorremos el subárbol y almacenamos los pesos de la rama izquierda en Acc1 y los pesos de la rama derecha en Acc2, sumamos ambos acumuladores y obtenemos el resultado en Res.

```
rama(tree(par(_,N),Left,Right),Acc1,Acc2,Res):-
      mod(N,s(s(0)),s(0)),
      rama(Left,Acc1,Acc2,Acc1),
      rama(Right,Acc1,Acc2,Acc2),
      plus(Acc1,Acc2,Res).
```

Consultas Realizadas para arbolBalanceadoPar/1

A continuación, realizamos algunas pruebas para comprobar que el predicado funciona correctamente. Le pasamos al predicado un árbol en el que la suma de los pesos de las ramas izquierda y derecha son iguales.

```
?-arbolBalanceadoPar(tree(par(c,s(0)),tree(par(e,s(s(0))),void,void),tre
e(par(s,s(s(0))),void,void))).

yes.
```

Las siguientes pruebas verifican que si le pasamos un árbol que no está balanceado, es decir que las suma de los pesos de las ramas derecha e izquierda no son iguales, el predicado funciona correctamente.

```
?-arbolBalanceadoPar(tree(par(h,s(0)),tree(par(o,s(s(0))),void,void),tre
e(par(y,s(0)),void,void))).

no.

?-
arbolBalanceadoPar(tree(par(h,s(0)),tree(par(o,s(s(s(0)))),void,void),tr
ee(par(y,s(s(s(s(0))))),void,void))).

no.
```

Segunda Etapa del Algoritmo

Predicado arbolAmplificado/2:

Dado un árbol, se incrementa el peso de los nodos en el número de veces que indica la raíz del árbol origen. Para ello, pasamos un árbol como parámetro de entrada en el que se

analizará por un lado el hijo derecho y por el otro el hijo izquierdo, llamamos a subtree/3 para amplificar el peso de los nodos. Una vez que termina subtree/3, devolvemos el árbol resultado con la última cláusula que realiza la unificación de AAmp con el nuevo árbol.

```
arbolAmplificado(void,_).
arbolAmplificado(tree(par(S,N),L,R),AAmp):-
      subtree(L,N,LAmp),
      subtree(R,N,RAmp),
      AAmp = tree(par(S,N),LAmp,RAmp).
```

```
subtree(L,N,LAmp) ^ subtree(R,N,RAmp) ^ AAmp = tree(par(S,N),LAmp,RAmp)
→
arbolAmplificado(tree(par(S,N),L,R),AAmp)
```

Predicado subtree/3

El siguiente predicado tiene como argumentos de entrada: un árbol y la raíz del árbol original. El argumento de salida es Stree que contendrá el árbol amplificado, manteniendo la raíz del árbol original.
Los Argumentos del predicado son:

- tree(par(S,N),L,R): árbol
- Root: Raíz del árbol
- STree: nuevo árbol

```
subtree(void,_,void).
subtree(tree(par(S,N),L,R),Root,STree):-
      subtree(L,Root,L1),
      subtree(R,Root,R1),
      times(N,Root,Res),
      STree = tree(par(S,Res),L1,R1).
```

Consultas Realizadas para arbolAmplificado/2

Probamos el predicado, llamándolo con el siguiente árbol, y podemos observar que el árbol amplificado es el original pero con sus pesos modificados, pues se incrementa, con respecto al peso original, en el número de veces que indica la raíz del árbol origen.

```
?-arbolAmplificado(tree(par(a,s(s(0))),tree(par(m,s(0)),void,void),tree(
par(l,s(s(0))),void,void)),AAmp).

AAmp =
tree(par(a,s(s(0))),tree(par(m,s(s(0))),void,void),tree(par(l,s(s(s(s(0)
))))),void,void)) ? ;

no.
```

Ahora, procedemos a pasarle al predicado un árbol amplificado, con peso en la raíz de s(0), por lo que dejaría el árbol igual.

```
?-
arbolAmplificado(tree(par(n,s(0)),tree(par(o,s(0)),void,void),void),AAmp
).

AAmp = tree(par(n,s(0)),tree(par(o,s(0)),void,void),void) ? ;

no.
```

Conclusiones

En conclusión, estamos muy contentos con este proyecto, ya que hemos aprendido bastante sobre la programación declarativa, y nos ha dado otro enfoque de ver la programación. Por otro lado hemos afianzado conocimientos en emacs y ciao, y hemos descubierto un potente ide, así que a pesar del pequeño fallo que nos da cells al hacer la recursión a la inversa, el hecho de que el resto de problemas los hemos solventado correctamente, es para estar satisfecho.