AALBORG UNIVERSITY

COMPUTER SCIENCE DEPARTMENT

SOFTWARE SYSTEMS ENGINEERING

# The Nim game

## THE MINI-PROJECT FOR PROGRAMMING PARADIGMS

Eglė Sasnauskaitė        Marius Mikučionis
Supervisor: Michael Böehlen

December 12, 2001

# Contents

# 1 Introduction

The mini-project covers one of two parts represented during the course of Programming Paradigms - Logical Programming. We found it interesting and wanted to get a better understanding about logical programming since it was quite a new programming paradigm for us. For that purpose we have chosen the NIM game. It's description and strategy are in the appendix A [1].

# 2  Predicates

For programming the game Prolog programming language was used. We had to deal with different kinds of methods that Prolog provides [1]:

- Declarative Prolog for querying facts and rules. In our program we got to use lists, recursive rules, etc.

- Procedural Prolog was used as well since we needed to deal with input and output, arithmetics, manipulate flow control, self modifying programs, etc.

We will present all predicates used in the following sections.

## 2.1  goal

A user starts the game by entering a command *goal*. The *goal* initiates and starts the game with *init* and *play* predicates:
*goal:– init(HeapList),*
    *play(HeapList).*

## 2.2  init

The *init* predicate asks the user to enter the initial configuration of heaps. The input must be a prolog-style list of numbers which indicate how many matches are in certain heap:
*init(HeapList):– write('Enter heaps configuration: '),*
        *read(HeapList).*

## 2.3  play

The *play* controls the game flow. Predicate consists of three sentences:

1. Congratulates the user if the configuration of heaps corresponds to the final state, i.e. if the maximum number of matches in heaps is 0:

   *play(HeapList):– listMax(HeapList, 0),*
           *write('You won! Congratulations!'), !, fail.*
   To distinguish the computer win or loss we end up here by *fail*ing, i.e. the interpreter will say 'No' at the end of the game if computer loses.

2. If the heap configuration is not final then the next sentence tries to make a move and remembers that move. After successful move the predicate checks whether it has reached the final state and ends the game if so:

> *play(HeapList):– tryMove(move(me, HeapList, NewHeapList)),*
> *listMax(NewHeapList, 0),*
> *write('I won! :)'), !.*

This time the interpreter will end up the game with saying 'Yes'.

3. If the final state of heaps was not reached, the computer makes the same move as it has already calculated, forgets about it, gives aportunity for the user to move and proceeds with *play*ing:

> *play(HeapList):– move(me, HeapList, MyHeapList),*
> *retract(move(me, HeapList, MyHeapList):-!),*
> *move(user, MyHeapList, UserHeapList),*
> *play(UserHeapList).*

   Predicate *play* uses several other predicates *listMax*, *tryMove* and *move*.

## 2.4   listMax

The *listMax* predicate finds the the maximum number of the matches in heap. It consists of three sentences:

1. If the list of heaps is empty then an error message is printed out on the screen:

   *listMax([ ], _ ):– write('There is no maximum in empty list!'), !, fail.*

2. If the list contains just one heap then obviously this heap contains the maximum number of matches:

   *listMax([Number], Number):– !.*

3. Else the list of heaps contains more than one heap. The last sentence compares the first and the second heap in the list and proceeds with listing the maximum number with leaving the larger heap in the list:

   *listMax( $[N1, N2 \mid Tail]$, Max):–  Larger is max(N1, N2),*
   *listMax( $[Larger \mid Tail]$ , Max).*

   Recursive rule is used to take heaps from the beginning of the list and compare it with the number of matches in the next heap. A heap which has more matches is placed in the beginning of the list and compared with the next till the end of the list. The heap which stays in the list has the maximum number of matches.

## 2.5   move

*Move* is a dynamic predicate which makes a move for the computer and for the user. It consists of two sentences for the computer and two for the user:

1. The sentence for the loosing strategy when heaps are given already in a state of (computer) lost. At first it tests whether the heap configuration gives the NIM sum 0, then finds the largest heap of matches, takes just one match from that heap and informs the user about the actions:

   *move(me, HeapList, Res):– testHeaps(HeapList),*
   *maxIndex(HeapList, MaxIndex),*
   *subtract(HeapList, MaxIndex, 1, Res),*
   *write('I take 1 match from '),*
   *write(MaxIndex), write('th heap'), !.*

   Note that taking just one match from the largest heap is the best strategy when computer is losing: it prolongs the game and the longer game lasts the bigger probability that the user will make wrong move, which means the greater probability for computer to take over the winning state.

2. If the configuration of heaps is good for computer to win, computer finds the suitable heap, takes the suitable amount of matches from it and informs the user about the move:

   *move(me, HeapList, Res):– findHeaps(HeapList, Sub, Index, Res),*
   *write('I take '), write(Sub),*
   *write(' matches from '), write(Index),*
   *write('th heap.'), !.*

3. The interface for the user to make a move. The sentence fails and the next sentence is invoked if the user enters wrong move values. If the user makes a legal move then a new configuration of heaps is returned to the *play* predicate:

4

*move(user, HeapList, Res):– write('You have heaps of matches: '),*
*write(HeapList),*
*write('Choose heap number from 1 to '),*
*length(HeapList, Length),*
*write(Length), write(': '), read(Index),*
*Index ≥ 1, Index ≤ Length,*
*nth1(Index, HeapList, Heap),*
*Heap > 0,*
*write('Choose amount of matches from 1 to '),*
*write(Heap), write(': '),*
*read(UserTake),*
*UserTake ≥ 1, UserTake ≤ Heap,*
*subtract(HeapList,Index,UserTake,Res),*
*write('So now the heaps are: '),*
*write(Res), !.*

The user performs a move in two steps:

(a) Chooses a an index of a heap. The heap index should be between 1 and the length of the list.

(b) Chooses a number of matches to subtract from the heap. The number of the matches to subtract should be more than 0 and less or equal to the number of matches in the chosen heap.

4. If the user enters anything wrong to make a legal move, the computer informs him/her about it and gives another chance to enter right values:

*move(user, HeapList, Res):– write('Illegal move! Try again.'),*
*move(user, HeapList, Res).*

The dynamic behaviour of this predicate is explained in the *tryMove* predicate description.

## 2.6 tryMove

*TryMove* inserts a rule in the beginning of the program. This predicate is used to remember the last *move* between two *play* predicate sentences, which would search for the same *move* twice if not remembered. The definition for this predicate is general and can be used for any other fact to remember:
*tryMove(P):– P, asserta((P:-!))*

## 2.7  testHeaps

*TestHeaps* is used for testing whether a list of numbers have Nim sum 0:
*testHeaps(HeapList):– prepareLists(HeapList, BinLists),*
*binsum(BinLists, HeapSum),*
*bin2num(HeapSum, 0).*

From the definition you may see, that the numbers are converted into binary digit lists, the binary sum calculated and the number corresponding to the sum is compared with 0.

## 2.8  prepareLists

*PrepareLists* prepares the lists of binary digits given the configuration of heaps. It converts numbers binary digit lists and patches them to have the same length. It is assumed that the greatest number in the heaps list corresponds to the longest binary digit list. So the predicate finds the greatest number in the heaps list (using *listMax*), calculates the binary digit list (using *num2bin*), gets the length of the binary digit list (using standard Prolog predicate *length*) and invokes another sentence of *prepareLists* which actually calculates all the binary digit lists and patches them one by one:
*prepareLists(NumList, ListOfBinLists):–*
*listMax(NumList, Max),*
*num2bin(Max, BinMax),*
*length(BinMax, MaxLength),*
*prepareLists(NumList, MaxLength, ListOfBinLists).*

*prepareLists([N | Nt], MaxLength, [BinList | Rt]):–*
*num2bin(N, BL),*
*patch(BL, MaxLength, BinList),*
*prepareLists(Nt, MaxLength, Rt)*

Since the last sentence are recursive, we need to specify the end of the recursion:
*prepareLists([ ], ⎯, [ ]):– !.*

## 2.9  num2bin

*Num2bin* converts a given number to a binary digits list. If the number is 0 then the answer is the list containing 0, otherwise the calculation is done by other sentences with extra arguments:

6

*num2bin(Number, [Number]):– Number=:=0, !.*

*num2bin(Number, BinList):– num2bin(Number, 1, BinList, _).*

*num2bin(Number, PowerOf2, $[BinDigit]$ , 1):–*
   *Number // PowerOf2 =:= 1,*
   *BinDigit is Number mod 2, !.*
*num2bin(Number, PowerOf2, $[BinDigit \mid Tail]$, BinIndex):–*
   *P1 is $PowerOf2 \cdot 2$,*
   *num2bin(Number, P1, Tail, B1),*
   *BinIndex is $B1 \cdot 2$,*
   *BinDigit is Number // BinIndex mod 2.*

The idea of these sophisticated sentences is to go down by recursion to the last (actually the first in the binary digits list) bit of number, avaluate the *PowerOf2* and then calculate the binary digit by returning from the recursive calls.

We exploit the well known conversion formula: $b_a = (N \; div \; 2^a) \; mod \; 2$, where $a = 0 \ldots n$ and the first binary digit of the operation is the last in the binary digit number.

- *Number* is a given number, in our case it is decimal number.

- *BinList* is a list of binary digits.

- *PowerOf2* is 2 powered by the bit index in list (counting from the end of the list). Each bit (beginning with the 0th) has a corresponding *PowerOf2* number value $2^n$, where $n$ is the index of bit.

- *binIndex* is a number which shows the position of a binary digit in the binary list (index).

At first we have to find out what is the biggest power for 2 to get the last binary digit in the list. It is done when   $Number // PowerOf2 =:= 1$   and then the last binary digit is   $BinDigit \; is \; Number \; mod \; 2.$   Then we go backwards and find other digits in the list increasing index number.

## 2.10 patch

*Patch* adds up zeros to the binary lists to have the same length lists. The predicate calculates the length of the list by standard Prolog function *length* and compares it with the given length. If the length of the list is equal or bigger than the given length, the patching is not processed:

*patch(BinList, Length, BinList):– length(BinList, Length), !.*

*patch(BinList, Length, _):– length(BinList, Ln), Ln > Length, !, fail.*

  *Ln* is the length of the list, *Length* is the given length. If the list is shorter than the given length then zeroes are patched in the front of the list till the given length is reached:
*patch(BinList, Length, Patched):– patch($[0 \mid BinList]$, Length, Patched).*

## 2.11 binsum

*Binsum* is another procedure in *testHeaps*. *binsum* sums up several binary digit lists of the same length into one binary digit list. The output is another list of binary digits:
*binsum([ ],[ ]):– !.*

*binsum([BinList],BinList):– !.*

*binsum($[X, Y \mid Tail]$, Z):– binplus(X,Y,X1),*
        *binsum($[X1 \mid Tail]$, Z).*

## 2.12 binPlus

*binsum* uses the plus operation for the binary lists: *binPlus*. *binplus* adds up two equal-length binary digit lists. The sums are transformed to binary digits using modulus division and the third list of binary digits is produced bit-by-bit:
*binplus($[X \mid Xt]$, $[Y \mid Yt]$, $[Z \mid Zt]$):– Z is $(X + Y)mod2$,*
        *binplus(Xt, Yt, Zt).*
The predicate is recursive and the condition for the end of recursion is empty lists in the input:
*binplus([ ], [ ], [ ]):– !.*

## 2.13 bin2num

*bin2num* is the last procedure in *testHeaps*. This procedure does an reverse operation than num2bin: it converts a list of binary digits to a number. If the list contains just one digit, the digit is taken as a number:

*bin2num([BinDigit], BinDigit, 1):– !.*

At first we find the last digit in the tail of the list using recursion. Last digit is multiplied by 2 with power 1 and this is the starting decimal number.

*bin2num([$BinDigit \mid Tail$], Number, PowerOf2):–*
        *bin2num(Tail, N, P1),*
        *PowerOf2 is $P1 \cdot 2$,*
        *Number is $N + PowerOf2 \cdot BinDigit$.*

Then previous digit is taken, multiplied by 2 with incremented number of power and added to the starting decimal number. The procedure is repeated until the first binary digit is reached and final decimal number is obtained.

The formula for converting from binary to decimal number is:

$Number = \Sigma b_a \cdot 2^a$

where $a = 0 \ldots n$ from the left to the right of the list, $b_i$ is a value of i-th bit. The binary list must be in this shape: $(b_n, b_{n-1}, \ldots, b_1, b_0)$

## 2.14    subtract

*subtract* subtracts a number from a number (heap) in the list with given index. Predicate has three input arguments and the output is the same-length list of the number where one of the numbers is smaller by the given subtrahend than before subtraction:

$subtract(+ListOfNums, +IndexOfNumInList, +Subtrahend, -ResList)$

The index number has to be smaller or equal to the length of the list. At first the certain number is found checking indexes from the beginning of the list till the end. When the necessary number becomes first in the list subtraction is processed.

*subtract(List, Index, _, _):– length(List, Length),*
                   *Index>Length,*
                   *write('Index is bigger than length of list!'), !, fail.*

*subtract([$Number \mid Nt$], 1, X, [$Y \mid Nt$]):– Y is Number-X.*

*subtract([$Number \mid Nt$], I, X, [$Number \mid Rt$]):– $I > 1$,*
                                 *I1 is I-1,*
                                 *subtract(Nt, I1, X, Rt).*

## 2.15    findHeaps

*findHeaps* consists of procedures that check the length of the list of heaps (procedure *length*). The length is used as a maximum limit for generating a number which would indicate an index of a certain heap.

*findHeaps(HeapList, Sub, Index, NewHeaps):–*
  *length(HeapList, ListLength),*
  *generate(ListLength, Index),*
  *nth1(Index, HeapList, Heap),*
  *generate(Heap, Sub),*
  *subtract(HeapList, Index, Sub, NewHeaps),*
  *testHeaps(NewHeaps).*

Procedure *nth1* finds out how many matches are there in the heap. In the next step the computer generates (procedure *generate*) a number of matches which the computer wants to subtract (procedure *subtract*) from the chosen heap. After subtraction the configuration of heaps is checked by procedure *testHeaps*. If the Nim sum is not equal to 0 then numbers (for a heap index and for number of matches) are re-generated until Nim sum is 0.

The logic of this predicate can be expressed in pseudocode of imperative programing:

*for Index = ListLength downto 1 do begin*
  *Heap = HeapList[Index];*
  *for Sub = Heap downto 1 do begin*
    *NewHeaps = HeapList;*
    *NewHeaps[Index] = HeapList[Index] - Sub;*
    *if testHeaps(NewHeaps) then return true;*
  *end;*
*end;*

## 2.16   generate

*generate* generates numbers from the given number to 1 in descending order. The computer generates numbers when it wants to chose a certain heap in the list and to choose certain amount of matches for subtraction. This procedure is similar to the loop "For . . . to" in other programming paradigms:

*generate(0, 0):– !, fail.*

*generate(Number, Number):– .*

*generate(Number, N):– N1 is $Number - 1$, generate(N1, N).*

# 3  Results

The program announces who is the winner. Controls the input of the user and announces the errors.

# 4  Conclusions

Programming the game was a good opportunity to learn Prolog. We followed the strategy which was introduced and described in the task. Recommendations for how to program in prolog were used:

- For every procedure we found out what facts we have and what can we use as an input and how to join the facts in the rule to get the output.

- Data objects and data structures were analyzed.

- Operations with list were performed.

- Recursive rules were used.

- Issues of procedural semantics were analyzed nd understood.

- Arithmetical operations were performed.

- Input output mechanism was used.

- Control flow manipulations using cuts were used.

- Mechanism of self modifying programs was introduced.

- Dynamic function included in the program.

- We used tracing mechanism to understand what steps the program performs while debuging.

# References

[1] Programming Paradigms Part 2: Logic Programming
http://www.cs.auc.dk/ normark/prog3-01/html/pp.html

[2] Roman Bartak Guide to Prolog Programming http://kti.ms.mff.cuni.cz/ bar-
tak/prolog/learning.html

# A   Nim (Logic Programming Paradigm)

Description

Nim starts with a group of heaps, each of which contains one or more matches.
Each player takes turns removing one or more matches from a single heap. It is
OK to remove and entire heap. The player who picks up the last match wins the
game. The standard configuration consists of three heaps, with three, five, and
seven matches, respectively. Each player picks at least one match from any one
heap during each turn.

Strategy

With multiple heaps and the possibility to remove as many matches as you
want from any one of the heaps, you need to compute a nim sum, that characterizes
the configuration of the game. The details are as follows:

- Express the number of matches in each heap as a binary number.

- Patch small binary numbers with '0's to the left so that all numbers have the
  same number of digits.

- Sum the binary numbers, but do not carry.

- Replace each digit in the sum with the remainder that results when the digit
  is divided by 2.

- This yields the nim sum.

- To win at nim, always make a move that leaves a configuration with a nim
  sum of 0. If you cannot do this, your opponent has the advantage, and you
  have to hope for a mistake in order to win.

  - If the configuration you are given has a nim sum not equal to 0, there
    is always a move that creates a new configuration with a nim sum of
    0.

  - If you are given a configuration that has a nim sum of 0, there is no
    move that will create a configuration that also has a nim sum of 0.

Here is an example of computing a nim sum. Assume you have three heaps, with 3, 7, and 11 matches, respectively.

- As binary numbers, these quantities are 11, 111, and 1011, respectively.

- Patching with '0's yields 0011, 0111, and 1011.

- Summing without carrying gives: $0011 + 0111 + 1011 = 1133$

- Taking the remainders after dividing each digit by 2 yields 1111, which is not equal to 0. Therefore, this configuration is a winning position for the person who is about to take a turn. You need to plan a move that creates a configuration with a nim sum of 0. This can be achieved by removing 7 matches from the pile that now has 11. We then get: $0011 + 0111 + 0100 = 0222$ The remainders after dividing by 2 gives 0000.

# B    The source code of the Nim game

```
% NUMBER TO LIST OF BINARY DIGITS
% dec2bin(+Number, -BinaryList)
num2bin(Number, [Number]):- Number=:=0, !.
num2bin(Number, BinList):- num2bin(Number, 1, BinList, _).
% dec2bin(+Number, +PowerOf2, -binlist, -binIndex)
num2bin(Number, PowerOf2, [BinDigit] , 1):-
        Number // PowerOf2 =:= 1,
        BinDigit is Number mod 2, !.
num2bin(Number, PowerOf2, [BinDigit|Tail], BinIndex):-
        P1 is PowerOf2 * 2,
        num2bin(Number, P1, Tail, B1),
        BinIndex is B1*2,
        BinDigit is Number // BinIndex mod 2.


% LIST OF BINARY DIGITS TO NUMBER
%bin2num(+BinList, -Number)
bin2num(X,Y):- bin2num(X,Y,_).
% bin2num(+BinList, -Number, -PowerOf2)
bin2num([BinDigit], BinDigit, 1):- !.
bin2num([BinDigit|Tail], Number, PowerOf2):-
        bin2num(Tail, N, P1),
        PowerOf2 is P1 * 2,
        Number is N + PowerOf2 * BinDigit.


% PATCH BINARY LIST WITH ZEROS TILL LENGTH
% patch(+BinList, +Length, -Patched)
patch(BinList, Length, BinList):- length(BinList, Length), !.
patch(BinList, Length, _):- length(BinList, Ln), Ln > Length, !, fail.
patch(BinList, Length, Patched):- patch([0|BinList], Length, Patched).


% FIND THE MAXIMUM VALUE OF NUMBER IN THE LIST
% listmax(+NumberList, -MaxNumber)
listMax([],_):-
        write('There is no maximum in empty list!'), !, fail.
listMax([Number],Number):- !.
listMax([N1,N2|Tail],Max):- Larger is max(N1,N2), listMax([Larger|Tail], Max).


% FIND THE INDEX OF MAXIMUM VALUE NUMBER IN THE LIST
% maxIndex(+NumberList, -MaxIndex)
maxIndex(NumberList, MaxIndex):-
        listMax(NumberList, Max),
        nth1(MaxIndex, NumberList, Max).


% THE BINARY PLUS OPERATION ON TWO LISTS OF BINARY DIGITS
% binplus(+BinList1, +BinList2, -SumBinLists)
binplus([],[],[]):- !.
binplus([X|Xt],[Y|Yt],[Z|Zt]):-
        Z is (X+Y) mod 2,
        binplus(Xt,Yt,Zt).


% THE BINARY SUM OF LISTS OF LISTS OF BINARY DIGITS
% binsum(+ListOfBinLists, -SumBinList)
binsum([],[]):- !.
binsum([BinList],BinList):- !.
binsum([X,Y|Tail], Z):-
        binplus(X,Y,X1),
        binsum([X1|Tail],Z).


% SUBTRACTS A NUMBER FROM SELECTED NUMBER IN THE NUMBER LIST
% subtract(+ListOfNumbers, +IndexOfNumberInList, +NumberToSubtract, -ResList)
subtract(List, Index, _, _):-
        length(List, Length),
        Index>Length,
        write('Index is bigger than length of list!\n'), !, fail.
% USE THIS FOR HINT: subtract([Number|Nt], 1, X, [Number-X|Nt]).
```

14

```
subtract([Number|Nt], 1, X, [Y|Nt]):- Y is Number-X.
subtract([Number|Nt], I, X, [Number|Rt]):-
        I > 1,
        I1 is I-1,
        subtract(Nt,I1,X,Rt).

% PREPARES BINARY DIGIT LISTS FROM LIST OF NUMBERS
% prepareLists(+ListOfNumbers, -ListOfUnifiedLengthBinLists)
prepareLists([],[]):- !.
prepareLists(NumList, ListOfBinLists):-
        listMax(NumList, Max),
        num2bin(Max, BinMax),
        length(BinMax, MaxLength),
        prepareLists(NumList, MaxLength, ListOfBinLists).
prepareLists([],_,[]):- !.
prepareLists([Number|Nt], MaxLength, [BinList|Rt]):-
        num2bin(Number, BL),
        patch(BL, MaxLength, BinList),
        prepareLists(Nt,MaxLength,Rt).

% GENERATES NUMBERS BETWEEN 1 AND MAX
% generate(+Max, -Number)
generate(0,0):- !, fail.
generate(Number, Number).
generate(Number, N):- N1 is Number-1, generate(N1,N).

% SUCCEEDS IFF BINARY SUM OF BINARY HEAPS IS 0
% testHeaps(+HeapList)
testHeaps(HeapList):-
        prepareLists(HeapList,BinLists),
        binsum(BinLists,HeapSum),
        bin2num(HeapSum, 0).

% THE LOOP FOR GENERATING AND TESTING THE SUBTRAHEND AND INDEX
% findHeaps(HeapList+, Sub-, NewHeaps-)
findHeaps(HeapList, Sub, Index, NewHeaps):-
        length(HeapList, ListLength),
        generate(ListLength, Index),
        nth1(Index, HeapList, Heap),
        generate(Heap, Sub),
        subtract(HeapList, Index, Sub, NewHeaps),
        testHeaps(NewHeaps).

% COMPUTER MOVE TRANSITION TO ANOTHER HEAPS CONFIGURATION
% move(+player, +HeapList, -ResultingHeapsAfterComputerMove)
:- dynamic move/3.
move(me, HeapList, Res):-
        testHeaps(HeapList),
        maxIndex(HeapList, MaxIndex),
        subtract(HeapList, MaxIndex, 1, Res),
        write('I take 1 match from '), write(MaxIndex),
        write('th heap\n'), !.
move(me, HeapList, Res):-
        findHeaps(HeapList, Sub, Index, Res),
        write('I take '), write(Sub),
        write(' matches from '), write(Index),
        write('th heap.\n'), !.

% USER MOVE SIMULATION: ASKS FROM WHICH HEAP AND HOW MUCH TO TAKE
move(user, HeapList, Res):-
        write('You have heaps of matches: '),
        write(HeapList), write('\n'),
        write('Choose heap number from 1 to '),
        length(HeapList, Length),
        write(Length),
```

```prolog
        write(': '), read(Index),
        Index >= 1, Index =< Length,
        nth1(Index, HeapList, Heap),
        Heap > 0,
        write('Choose number of matches to take from 1 to '),
        write(Heap), write(': '),
        read(UserTake),
        UserTake >= 1, UserTake =< Heap,
        subtract(HeapList,Index,UserTake,Res),
        write('So now the heaps are: '),
        write(Res),
        write('\n'), !.
move(user, HeapList, Res):-
        write('Illegal move! Try again.\n'),
        move(user, HeapList, Res).

tryMove(P):- P, asserta((P:-!)).

% ASKS THE USER FOR THE INITIAL HEAPS CONFIGURATION
% init(-HeapList)
init(HeapList):-
        write('Enter number of matches in heaps [N1, N2, N3]: '),
        read(HeapList).

% PLAY CONTROL: PLAY UNTIL SOMEONE WINS
% play(+HeapList)
play(HeapList):-
        listMax(HeapList, 0),
        write('You won :(\n Congratulations!\n'), !, fail.
play(HeapList):-
        tryMove(move(me, HeapList, NewHeapList)),
        listMax(NewHeapList, 0),
        write('I won! :)\n'), !.
play(HeapList):-
        move(me, HeapList, MyHeapList),
        retract(move(me, HeapList, MyHeapList):-!),
        move(user, MyHeapList, UserHeapList),
        play(UserHeapList).

% OUR GOAL IS TO INITIALIZE THE GAME AND PLAY :)
% goal
goal:-
        init(HeapList),
        play(HeapList).
```