NimLearn: A Learning Nim Player

Stuart C. Shapiro
Department of Computer Science and Engineering
State University of New York at Buffalo
226 Bell Hall
Buffalo, NY 14260-2000
`shapiro@cse.buffalo.edu`

December 23, 1999

1 Introduction

Nimlearn is a program that plays Nim interactively, and improves its play with experience. This paper describes the game of Nim, the techniques used in this program to play the game and to learn to improve its play, and some interesting implementation techniques used in the program.

2 The Game

Nim is a class of two-person, zero-sum, perfect-information games that have the following characteristics in common.

- There are one or more piles of pieces.
- The two players take turns removing some pieces from any one of the piles.
- Sometimes there is an additional constraint on the number of pieces that can be taken on any one move.
- The player who takes the last piece wins in some versions of the game, and loses in other versions.

The name "Nim" comes from the imperative form of the German word "nehmen", meaning "to take."

In the version of Nim played by this program, there is only one pile, but no more than `MAX` pieces may be taken in any one turn, and the player who takes the last piece loses.

The program will play a series of games against a person. At the beginning of each game, the person is asked to specify:

1. The value of `MAX` for the game.
2. How many pieces the pile should start with.
3. Whether (s)he wants to go first.

Here is an example of a game played by the program.

```
USER(1): :ld nimlearn
; Fast loading /projects1/shapiro/AIclass/nimlearn.fasl
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  3
How many shall we start with?  10
Do you want to go first?  no
I take 3, leaving 7
How many do you take?  1
That leaves 6.
I take 3, leaving 3
How many do you take?  2
That leaves 1.
I take 1, leaving 0
You win.

Do you want to play another game?  no
That was fun!  Come back another time.

USER(2):
```

3 Playing Strategy

All versions of Nim have been solved, and there is always a winning strategy for one of the players. For example, in the version of Nim played by this program, a winning strategy is to leave your opponent with $1 + i(\texttt{MAX} + 1)$ pieces, for any value of i. Thus, for a given value of `MAX`, whether the first or the second player has a forced win depends entirely on the size of the initial pile. However, a program that always played the winning strategy would not be interesting to play against, and, anyway, the purpose of this program is to demonstrate a particular learning technique.

The program bases its play on the entries of a two dimensional array `*tree*`, where `*tree*`$[i, j]$ indicates what should be played when there are j pieces left in a game for which `MAX` $= i$. The possible values of `*tree*`$[i, j]$, and their meanings are:

`NIL`: There is no information about this game state.

`(WIN . `x`)`: If it is your turn to move in a game for which `MAX` $= i$, and there are j pieces left, you can win by taking x pieces.

`(LOSE . 1)`: If it is your turn to move in a game for which `MAX` $= i$, and there are j pieces left, you will lose if your opponent makes no mistakes, so you may as well take just 1 piece to give your opponent the maximal chance to make a mistake.

The first time the program plays a game for which `MAX` $= i$, it sets `*tree*`$[i$`,0]` to `(WIN . 0)`, and `*tree*`$[i, j]$, $j > 0$, to `NIL`. The program adds non-`NIL` entries to the array as it plays, thus improving its future playing ability in games with the same `MAX`. The program does not transfer its learning from games with one value of `MAX` to games with different values of `MAX`.

The playing and learning strategy may best be understood by examining the function `I-play`, that chooses a play for the program, and learns when possible.

```
(defun I-play ()
  "Chooses a play for the program, makes the play, and learns when possible."
  (declare
   (special N    ; The number of pieces left in the pile
            MAX  ; The maximum number of pieces that may be taken in one turn.
            ))
  (let ((mymax (min N MAX))     ; The maximum number of pieces
        )                       ; that may be taken in this turn.
    (cond ((zerop N)              ; If there are 0 pieces left,
           (format t "~&I win!~%~%") ; the program wins,
           (throw 'gameover nil))  ; and the game is over.
          ((or (winning-state N) (losing-state N))
           ;; If this state is understood,
           ;; take the indicated number of pieces.
           (Iplay (play-of-state N)))
          (t (or                ; Otherwise, win if you can,
              ;; (winning-play N mymax) ; See this function for why it's not needed
              ;; but if you can't win, try to avoid losing,
              (non-losing-play N mymax)
              ;; but play a losing move if you have to.
              (losing-play N MAX))))))

(defun winning-play (N max)
  "If can make a winning play, make the biggest one, and return T;
   Else return NIL."
  ;; This function is included in this paper for completeness,
  ;; but it is not needed, because every winning state it will find is
  ;; already marked as a winning state.  That this is true follows from the
  ;; fact that the only place a losing state is recorded is in the function
  ;; losing-play, and that function also records all winning states that lead
  ;; to the losing state.
  (loop for x from max downto 1 do
        ;; If any legal move leads to a losing state,
        (when (losing-state (- N x))
          ;; note that this is a winning move
          ;; for future reference,
          (record-state N 'win x)
          ;; and make the move,
          (Iplay x)
          ;; and return T.
          (return T))))
```

```
(defun non-losing-play (N max)
  "If can make a play, not known to be a losing play,
   make it and return T;
   Else return NIL."
  (loop for x from max downto 1 do
        ;; Find the biggest move not known to be a losing move
        (when (unknown-state (- N x))
          ;; make it
          (Iplay x)
          ;; and return T.
          (return T))))

(defun losing-play (N MAX)
  "Have to play a losing play, so just take 1
   and mark states that lead here as winning states."
  ;; record that this is a losing state
  ;; for future reference
  (record-state N 'lose 1)
  (loop for x from 1 to MAX do
        ;; record all winning states
        ;; that can get here in one move
        (record-state (+ N x) 'win x))
  (Iplay 1))
```

Notice that when `*tree*[3,`i`]` is `NIL` the program takes the maximum 3 pieces until it reaches the area of the game where it is experienced, and that when it knows it will lose, it takes only 1 piece to give its opponent a chance to make a mistake.

The interaction below immediately follows the game shown above. (Text starting with a semicolon is a comment added to this paper.)

```
USER(2): :pa nimlearn
NIMLEARN(3): (aref *tree* 3)
#((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) NIL NIL NIL NIL NIL ...)
NIMLEARN(4): (play)
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  3
How many shall we start with?  10
Do you want to go first?  no
I take 3, leaving 7          ; because *tree*[3,10] is NIL
How many do you take?  2
That leaves 5.
I take 1, leaving 4          ; because every move leads to a winning state
How many do you take?  3
That leaves 1.
I take 1, leaving 0
You win.

Do you want to play another game?  no
That was fun!  Come back another time.

NIMLEARN(5):
```

After that game there are more non-NIL entries in `*tree*[3,*]`

```
NIMLEARN(5): (aref *tree* 3)
#((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) (LOSE . 1) (WIN . 1)
  (WIN . 2) (WIN . 3) NIL ...)
NIMLEARN(6):
```

and now the program can win as soon as it reaches a state it knows is a winning one:

```
NIMLEARN(6): (play)
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  3
How many shall we start with?  10
Do you want to go first?  yes
How many do you take?  3
That leaves 7.
I take 2, leaving 5
How many do you take?  1
That leaves 4.
I take 3, leaving 1
How many do you take?  1
That leaves 0.
I win!
```

4 A Game Tree Analysis

The one-dimensional array `*tree*[3,*]` can be viewed as a representation of the game tree, shown in Figure 1, for the game of Nim with `MAX = 3`. Now the learning technique can be seen as the gradual filling in of the values of the nodes of this tree according to the minimax rule: a node all of whose successors represent winning states represents a losing state, and a node any one of whose successors represents a losing state represents a winning state, and the winning move is the one that leads to that losing state.

5 Interesting Implementation Techniques

5.1 Allowing for Various Values of MAX

The two-dimensional array `*tree*` is implemented as a one-dimensional array, each of whose elements is either `NIL` or a one-dimensional array. The elements of `*tree*` are initialized as `NIL`, and when a new game is started, if `(aref *tree* MAX)` is `NIL`, it is set to a new array with `newtree`:

```
(defun newtree (MAX N)
  "Creates a new array indexed 0..N
   whose 0th element is marked as a winning state;
   It is made the value of *tree*[MAX];
   and returned."
  (declare (special *tree*))
  (let ((state (make-array (1+ N) :adjustable t)))
    (declare (special state))
    (setf (aref *tree* MAX) state)
    (record-state 0 'win 0)
    state))
```

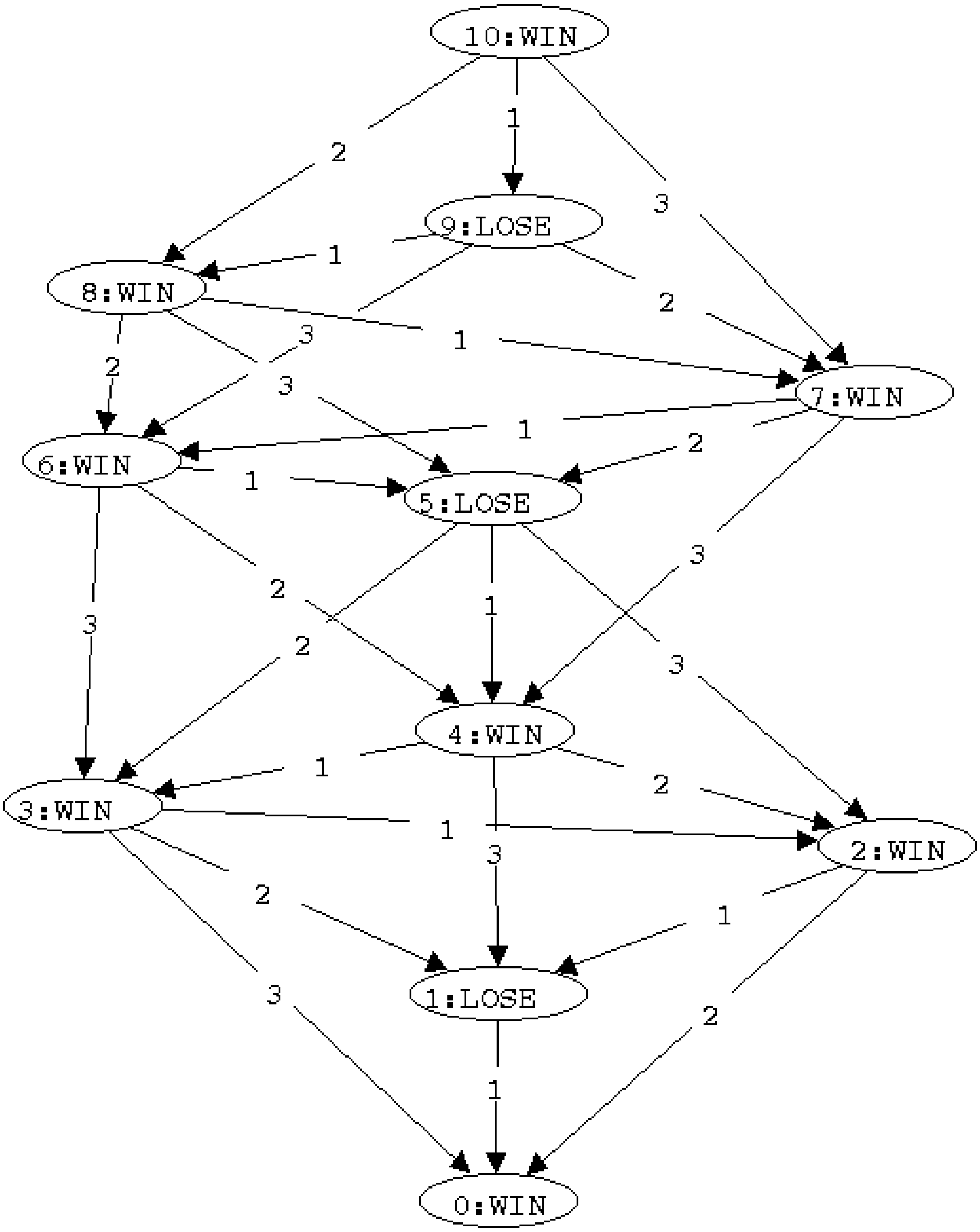

Figure 1: A game tree (actually DAG) for NIM with `MAX` = 3. The states are shown as i : *STATUS*, where i is the number of pieces left and *STATUS* is `WIN` for a winning state and `LOSE` for a losing state. The arcs are labelled with the number of pieces to be taken. To win from a winning state, make the move that goes to a losing state.

This can be seen in the following session.

```
NIMLEARN(7): (aref *tree* 4)
NIL
NIMLEARN(8): (play)
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  4
How many shall we start with?  10
Do you want to go first?  no
I take 4, leaving 6
How many do you take?  4
That leaves 2.
I take 1, leaving 1
How many do you take?  1
That leaves 0.
I win!

Do you want to play another game?  no
That was fun!  Come back another time.

NIMLEARN(9): (aref *tree* 4)
#((WIN . 0) NIL NIL NIL NIL NIL NIL NIL NIL NIL ...)
```

5.2 Allowing for Various Initial Pile Sizes

When `newtree` calls `make-array`, the keyword argument `:adjustable` is set to `T`, meaning that the array can be enlarged dynamically if need be. This is done if the opponent ever wants to start a game with more pieces than the original game at this `MAX`. The function `check-length` adjusts an array if need be.

```
(defun check-length (N array)
  "If the array has fewer than [0..N] elements, adjust it.
   Return array, possibly adjusted."
  (if (< N (array-dimension array 0)) array
    (adjust-array array (1+ N))))
```

The adjustment may be seen in this session.

```
NIMLEARN(10): (setf top-level:*print-length* 25)
25
NIMLEARN(11): (aref *tree* 3)
#((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) (LOSE . 1) (WIN . 1)
  (WIN . 2) (WIN . 3) NIL NIL)
NIMLEARN(12): (play)
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  3
How many shall we start with?  15
Do you want to go first?  no
I take 3, leaving 12
How many do you take?  3
That leaves 9.
I take 1, leaving 8
How many do you take?  3
That leaves 5.
```

```
I take 1, leaving 4
How many do you take?  3
That leaves 1.
I take 1, leaving 0
You win.

Do you want to play another game?  no
That was fun!  Come back another time.

NIMLEARN(13): (aref *tree* 3)
#((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) (LOSE . 1) (WIN . 1)
  (WIN . 2) (WIN . 3) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) NIL NIL NIL)
```

The array `*tree*`, itself, is initialized to have `10` elements, and is adjustable.

```
(defvar *tree* (make-array 10 :adjustable t)
  "tree[i] is the game tree for Nim with a MAX take of i")
```

5.3 Preserving What's Learned across Runs

The top-level function, `play`, takes a file name as an optional argument. If this argument is supplied, the value of `*tree*` is read from that file at the beginning of play, and written to it at the end. If the file does not exist at the beginning of play, the default value of `*tree*` is used, but if it does, when the new version is written at the end of play, the old version is saved as a back-up. This behavior can be seen in the following session, begun in a new COMMON LISP enviornment.

```
NIMLEARN(4): (shell "cat nim-memory")
cat: nim-memory: No such file or directory
1

NIMLEARN(5): *tree*
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)

NIMLEARN(6): (play "nim-memory")
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  3
How many shall we start with?  10
Do you want to go first?  no
I take 3, leaving 7
How many do you take?  2
That leaves 5.
I take 3, leaving 2
How many do you take?  1
That leaves 1.
I take 1, leaving 0
You win.

Do you want to play another game?  no
That was fun!  Come back another time.
```

```
NIMLEARN(7): (shell "cat nim-memory")
#(NIL NIL NIL
  #((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) NIL NIL NIL NIL NIL
    NIL)
  NIL NIL NIL NIL NIL NIL)0

NIMLEARN(8): (play "nim-memory")
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  3
How many shall we start with?  10
Do you want to go first?  no
I take 3, leaving 7
How many do you take?  2
That leaves 5.
I take 1, leaving 4
How many do you take?  3
That leaves 1.
I take 1, leaving 0
You win.

Do you want to play another game?  no
That was fun!  Come back another time.

NIMLEARN(9): (shell "cat nim-memory")
#(NIL NIL NIL
  #((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) (LOSE . 1) (WIN . 1)
    (WIN . 2) (WIN . 3) NIL NIL)
  NIL NIL NIL NIL NIL NIL)0

NIMLEARN(10): (shell "cat nim-memory.bak")
#(NIL NIL NIL
  #((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3) NIL NIL NIL NIL NIL
    NIL)
  NIL NIL NIL NIL NIL NIL)0
```

The functions that read and write the memory file are shown here:

```
(defun initialize-tree (memory)
  "*tree* is initialized from the memory file if that file exists,
   otherwise it is left as is."
  (with-open-file
      (treestream memory :direction :input :if-does-not-exist nil)
    (declare (special *tree*))
    (when (streamp treestream)
      (setf *tree* (read treestream)))))

(defun output-tree (memory)
  "*tree* is written out to the memory file."
  (with-open-file
      (treestream memory :direction :output :if-exists :rename)
    (declare (special *tree*))
    (format treestream "~S" *tree*)))
```

5.4 Playing a Series of Games

The function `play` will play a series of games, allowing the opponent to pick a new `MAX` and size of starting pile for each game. Its behavior can be seen in the following quick series of games.

```
NIMLEARN(12): (play)
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  4
How many shall we start with?  10
Do you want to go first?  yes
How many do you take?  4
That leaves 6.
I take 4, leaving 2
How many do you take?  1
That leaves 1.
I take 1, leaving 0
You win.

Do you want to play another game?  yes
What shall be the maximum to take at one time?  4
How many shall we start with?  10
Do you want to go first?  no
I take 4, leaving 6
How many do you take?  1
That leaves 5.
I take 4, leaving 1
How many do you take?  1
That leaves 0.
I win!

Do you want to play another game?  no
That was fun!  Come back another time.
```

5.5 Use of Exceptional Exits

To simplify the program structure, the function that plays a single game, `play-a-game`, is an infinite loop that alternates between calling the functions `I-play` for having the program play and `opp-play` for getting the opponent's play. When the pile is emptied, an inner function terminates the game by throwing control to a catch that is wrapped around the call to `play-a-game`. The top-level organization can be seen in the functions `play` and `play-a-game`.

```
(defun play (&optional memory)
  "Play a series of games.
   If memory is given, it must be a file name;
   the initial *tree* is read from there, and the final *tree* is written there.
   If the memory file is initially empty,
   *tree* is initialized as usual, and the final *tree* is still written there."
  (declare (special *tree*))
  (when memory (initialize-tree memory))
  (format t "~&Let's play Nim.~%Whoever takes the last piece loses.~%")
  (loop (let* ((MAX (read-int "What shall be the maximum to take at one time?"))
               (N (read-int "How many shall we start with?"))
               (*tree* (check-length MAX *tree*)))
          (catch 'gameover
            (play-a-game MAX N (if (aref *tree* MAX) (check-length N (aref *tree* MAX))
                                  (newtree MAX N))))
          (unless (yes-or-no-p "~&Do you want to play another game?  ")
            (return))))
  (when memory (output-tree memory))
  (format t "~&That was fun!  Come back another time.~%~%")
  (values))

(defun play-a-game (MAX N state)
  "Play a game, with MAX the maximum that can be taken in
   one turn, starting with N pieces.
   State is the game tree."
  (declare (special MAX N state))
  (if (y-or-n-p "~&Do you want to go first?  ")
      (loop
        (opp-play)
        (I-play))
    (loop
      (I-play)
      (opp-play))))
```

The throws can be seen in the functions `I-play`, shown above, and `opp-play`, shown here:

```
(defun opp-play ()
  "Gets the opponent's move, and makes it."
  (declare (special
            N    ; The number of pieces left in the pile
            MAX  ; The maximum number of pieces that may be taken in one turn.
            ))
  (cond ((zerop N)
         (format t "~&You win.~%")
         (throw 'gameover nil))
        (t (format t "~&That leaves ~A.~%"
                   (setf N
                     (- N (read-int "How many do you take?" (min N MAX))))))))
```

The effect of the throws can be seen by tracing `play`, `play-a-game`, `I-play`, and `opp-play`.

```
NIMLEARN(12): (trace play play-a-game I-play opp-play)
(OPP-PLAY I-PLAY PLAY-A-GAME PLAY)
NIMLEARN(13): (play)
 0: (PLAY)
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  5
How many shall we start with?  10
   1: (PLAY-A-GAME 5 10 #((WIN . 0) NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL))
Do you want to go first?  y
     2: (OPP-PLAY)
How many do you take?  3
That leaves 7.
     2: returned NIL
     2: (I-PLAY)
I take 5, leaving 2
     2: returned T
     2: (OPP-PLAY)
How many do you take?  1
That leaves 1.
     2: returned NIL
     2: (I-PLAY)
I take 1, leaving 0
     2: returned NIL
     2: (OPP-PLAY)
You win.

     2: returned-by-throwing NIL
   1: returned-by-throwing NIL
Do you want to play another game?  yes
What shall be the maximum to take at one time?  5
How many shall we start with?  10
   1: (PLAY-A-GAME 5 10
                   #((WIN . 0) (LOSE . 1) (WIN . 1) (WIN . 2) (WIN . 3)
                     (WIN . 4) (WIN . 5) NIL NIL NIL NIL))
Do you want to go first?  no
     2: (I-PLAY)
I take 3, leaving 7
     2: returned T
     2: (OPP-PLAY)
How many do you take?  1
That leaves 6.
     2: returned NIL
     2: (I-PLAY)
I take 5, leaving 1
     2: returned NIL
     2: (OPP-PLAY)
How many do you take?  1
That leaves 0.
     2: returned NIL
     2: (I-PLAY)
I win!
```

```
     2: returned-by-throwing NIL
   1: returned-by-throwing NIL
Do you want to play another game?  no
That was fun!  Come back another time.

 0: returned
```

Note the instances of "returned-by-throwing NIL".

Also notice that the last form in the definition of play is (values), which causes the function to terminate without returning a final value.

5.6 Checking Validity of Input

All input from the opponent is checked for validity, either by the COMMON LISP functions y-or-n-p and yes-or-no-p, or by the function read-int:

```
(defun read-int (prompt &optional max)
  "Reads and returns a number that must be a positive integer.
   If max is present, the number must not exceed max.
   Prompt is the prompt."
  (format t "~&~A  " prompt)
  (loop
    (let ((x (read)))
      (when (and (integerp x)
                 (plusp x)
                 (or (null max) (<= x max)))
        (return-from read-int x))
      (format t "~&That must be a positive integer")
      (when max (format t " not greater than ~A" max))
      (format t ".~%Try again:  "))))
```

This validity checking can be seen in the following session:

```
NIMLEARN(14): (play)
Let's play Nim.
Whoever takes the last piece loses.
What shall be the maximum to take at one time?  whatever
That must be a positive integer.
Try again:  -3
That must be a positive integer.
Try again:  3
How many shall we start with?  dont-care
That must be a positive integer.
Try again:  0
That must be a positive integer.
Try again:  12
Do you want to go first?  sure
Type "y" for yes or "n" for no.
Do you want to go first?  y
How many do you take?  11
That must be a positive integer not greater than 3.
Try again:  3
That leaves 9.
I take 1, leaving 8
```

```
How many do you take?  1
That leaves 7.
I take 2, leaving 5
How many do you take?  0
That must be a positive integer not greater than 3.
Try again:  3
That leaves 2.
I take 1, leaving 1
How many do you take?  none
That must be a positive integer not greater than 1.
Try again:  1
That leaves 0.
I win!

Do you want to play another game?  uh-uh
Type "yes" for yes or "no" for no.
Do you want to play another game?  no
That was fun!  Come back another time.
```

6 Summary

Nimlearn is a program that plays Nim interactively, and improves its play with experience. The improvement comes from a learning algorithm in which the game tree is gradually filled-in from the leaves up as the states are encountered during play. What the program learns may be saved across runs.

The program is flexible enough to play games of Nim with any maximum number of pieces to be taken in any one turn, and any size of initial pile.

Appendix: The Program

```
;;; NimLearn: A Learning Nim Player
;;; by Stuart C. Shapiro
;;; begun January 8, 1996
;;; for CS4/572
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Packaging Forms
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defpackage Nimlearn
  (:use common-lisp excl)
  (:export play))

(in-package Nimlearn)

(defvar *tree* (make-array 10 :adjustable t)
  "tree[i] is the game tree for Nim with a MAX take of i")
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Top-Level Game Player
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun play (&optional memory)
  "Play a series of games.
   If memory is given, it must be a file name;
   the initial *tree* is read from there, and the final *tree* is written there.
   If the memory file is initially empty,
   *tree* is initialized as usual, and the final *tree* is still written there."
  (declare (special *tree*))
  (when memory (initialize-tree memory))
  (format t "~&Let's play Nim.~%Whoever takes the last piece loses.~%")
  (loop (let* ((MAX (read-int "What shall be the maximum to take at one time?"))
               (N (read-int "How many shall we start with?"))
               (*tree* (check-length MAX *tree*)))
          (catch 'gameover
            (play-a-game MAX N (if (aref *tree* MAX) (check-length N (aref *tree* MAX))
                                 (newtree MAX N))))
          (unless (yes-or-no-p "~&Do you want to play another game?  ")
            (return))))
  (when memory (output-tree memory))
  (format t "~&That was fun!  Come back another time.~%~%")
  (values))

(defun play-a-game (MAX N state)
  "Play a game, with MAX the maximum that can be taken in
   one turn, starting with N pieces.
   State is the game tree."
  (declare (special MAX N state))
  (if (y-or-n-p "~&Do you want to go first?  ")
      (loop (opp-play) (I-play))
    (loop (I-play) (opp-play))))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Playing Functions for Program Player
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun I-play ()
  "Chooses a play for the program,
   makes the play,
   and learns when possible."
  (declare
   (special N    ; The number of pieces left in the pile
            MAX  ; The maximum number of pieces that may be taken in one turn.
            ))
  (let ((mymax (min N MAX))     ; The maximum number of pieces
                                ; that may be taken in this turn.
        )
    (cond ((zerop N)               ; If there are 0 pieces left,
           (format t "~&I win!~%~%") ; the program wins,
           (throw 'gameover nil))  ; and the game is over.
          ((or (winning-state N) (losing-state N))
           ;; If this state is understood,
           ;; take the indicated number of pieces.
           (Iplay (play-of-state N)))
          (t (or ; Otherwise,
              ;; try to avoid losing,
              (non-losing-play N mymax)
              ;; but play a losing move if you have to.
              (losing-play N MAX))))))

(defun non-losing-play (N max)
  "If can make a play, not known to be a losing play,
   make it and return T;
   Else return NIL."
  (loop for x from max downto 1 do
        ;; Find the biggest move not known to be a losing move
        (when (unknown-state (- N x))
          ;; make it
          (Iplay x)
          ;; and return T.
          (return T))))

(defun losing-play (N MAX)
  "Have to play a losing play, so just take 1
   and mark states that lead here as winning states."
  ;; record that this is a losing state
  ;; for future reference
  (record-state N 'lose 1)
  (loop for x from 1 to MAX do
        ;; record all winning states
        ;; that can get here in one move
        (record-state (+ N x) 'win x))
  (Iplay 1))
```

```lisp
(defun Iplay (x)
  (declare (special N))  ; The number of pieces left in the pile
  (format t "~&I take ~A, leaving ~A~%"
          x (setf N (- N x))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Function to Get Opponent's Play
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun opp-play ()
  "Gets the opponent's move, and makes it."
  (declare (special
            N   ; The number of pieces left in the pile
            MAX ; The maximum number of pieces that may be taken in one turn.
            ))
  (cond ((zerop N)
         (format t "~&You win.~%~%")
         (throw 'gameover nil))
        (t (format t "~&That leaves ~A.~%"
                   (setf N
                     (- N
                        (read-int "How many do you take?" (min N MAX))))))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Game Tree Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun initialize-tree (memory)
  "*tree* is initialized from the memory file if that file exists,
   otherwise it is left as is."
  (with-open-file
      (treestream memory :direction :input :if-does-not-exist nil)
    (declare (special *tree*))
    (when (streamp treestream)
      (setf *tree* (read treestream)))))

(defun output-tree (memory)
  "*tree* is written out to the memory file."
  (with-open-file
      (treestream memory :direction :output :if-exists :rename)
    (declare (special *tree*))
    (format treestream "~S" *tree*)))

(defun newtree (MAX N)
  "Creates a new array indexed 0..N
   whose 0th element is marked as a winning state;
   It is made the value of *tree*[MAX];
   and returned."
  (declare (special *tree*))
  (let ((state (make-array (1+ N) :adjustable t)))
    (declare (special state))
    (setf (aref *tree* MAX) state)
    (record-state 0 'win 0)
    state))
```

```lisp
(defun check-length (N array)
  "If the array has fewer than [0..N] elements, adjust it.
   Return array, possibly adjusted."
  (if (< N (array-dimension array 0)) array
    (adjust-array array (1+ N))))

(defun winning-state (x)
  "Returns T if state[x] is a winning state; NIL otherwise."
  (declare (special state))        ; The one-dimensional array *tree*[MAX,*].
  (eql (car (aref state x)) 'WIN))

(defun losing-state (x)
  "Returns T if state[x] is a losing state; NIL otherwise."
  (declare (special state))        ; The one-dimensional array *tree*[MAX,*].
  (eql (car (aref state x)) 'LOSE))

(defun unknown-state (x)
  "Returns T if state[x] is not known to be either a winning or a losing state;
   NIL otherwise."
  (declare (special state))        ; The one-dimensional array *tree*[MAX,*].
  (null (aref state x)))

(defun play-of-state (x)
  "Returns the play recorded in state[x]."
  (declare (special state))        ; The one-dimensional array *tree*[MAX,*].
  (cdr (aref state x)))

(defun record-state (x status play)
  "Records that state[x] has the given status (either WIN of LOSE),
   and that the correct play from here is play."
  (declare (special state))        ; The one-dimensional array *tree*[MAX,*].
  (setf state (check-length x state)
        (aref state x) (cons status play)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Function for Validity-Checking Input
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun read-int (prompt &optional max)
  "Reads and returns a number that must be a positive integer.
   If max is present, the number must not exceed max.
   Prompt is the prompt."
  (format t "~&~A  " prompt)
  (loop (let ((x (read)))
          (when (and (integerp x)
                     (plusp x)
                     (or (null max) (<= x max)))
            (return-from read-int x))
          (format t "~&That must be a positive integer")
          (when max (format t " not greater than ~A" max))
          (format t ".~%Try again:  "))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Final form causing play to begin when the file is loaded.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(play)
```