17

Applications

This chapter illustrates program structure via two examples: the first uses a modular model; the second, an object model.

The first application provides a set of parametric modules for two player games. A functor implements the minimax-$\alpha\beta$ algorithm for the evaluation of a search tree. A second functor allows modifying the human/machine interface for the game. These parametric modules are then applied to two games: a vertical tic-tac-toe game, and another involving the construction of mystic ley-lines.

The second application constructs a world where robots evolve. The world and robots are structured as classes. The different behaviors of robots are obtained by inheritance from a common abstract class. It is then easy to define new behaviors. There, too, the human/machine interface may be modified.

Each of the applications, in its structure, contains reusable components. It is easy to construct a new two player game with different rules that uses the same base classes. Similarly, the general mechanism for the motion of robots in a world may be applied to new types of robots.

Two Player Games

The application presented in this section pursues two objectives. On the one hand, it seeks to resolve problems related to the complexity in searching state spaces, as well as showing that Objective Caml provides useful tools for dealing with symbolic applications. On the other hand, it also explores the benefits of using parametric modules to define a generic scheme for constructing two player games, providing the ability to factor out one part of the search, and making it easy to customize components such as functions for evaluating or displaying a game position.

We first present the problem of games involving two players, then describe the *minimax-$\alpha\beta$* algorithm which provides an efficient search of the tree of possible moves. We present a parametric model for two player games. Then, we apply these functors to implement two games: "Connect Four" (a vertical tic-tac-toe), and Stonehenge (a game that involves constructing ley-lines).

The Problem of Two Player Games

Games involving two players represent one of the classic applications of symbolic programming and provide a good example of problem solving for at least two reasons:

- The large number of solutions to be analyzed to obtain the best possible move necessitates using methods other than brute force.
  For instance, in the game of chess, the number of possible moves typically is around 30, and a game often involves around 40 moves per player. This would require a search tree of around 30^{80} positions just to explore the complete tree for one player.
- The quality of a solution is easily verifiable. In particular, it is possible to test the quality of a proposed solution from one program by comparing it to that of another.

First, assume that we are able to explore the total list of all possible moves, given, as a starting point, a specific legal game position. Such a program will require a function to generate legal moves based on a starting position, as well as a function to evaluate some "score" for each resulting position. The evaluation function must give a maximum score to a winning position, and a minimal score to a losing position. After picking an initial position, one may then construct a tree of all possible variations, where each node corresponds to a position, the adjacent siblings are obtained by having played a move and with leaves having positions indicating winning, losing, or null results. Once the tree is constructed, its exploration permits determining if there exists a route leading to victory, or a null position, failing that. The shortest path may then be chosen to attain the desired goal.

As the overall size of such a tree is generally too large for it to be fully represented, it is typically necessary to limit what portions of the tree are constructed. A first strategy is to limit the "depth" of the search, that is, the number of moves and responses that are to be evaluated. One thus reduces the breadth of the tree as well as its height. In such cases, leaf nodes will seldom be found until nearly the end of the game.

On the other hand, we may try to limit the number of moves selected for additional evaluation. For this, we try to avoid evaluating any but the most favorable moves, and start by examining the moves that appear to be the very best. This immediately eliminates entire branches of the tree. This leads to the *minimax* $\alpha\beta$ algorithm presented in the next subsection.

Minimax $\alpha\beta$

We present the *minimax* search and describe a variant optimized using $\alpha\beta$ cuts. The implementation of this algorithm uses a parametric module, `FAlphabeta` along with a representation of the game and its evaluation function. We distinguish between the two players by naming them A and B.

Minimax

The *minimax* algorithm is a depth-first search algorithm with a limit on the depth to which search is done. It requires:

- a function to generate legal moves based on a position, and
- a function to evaluate a game position.

Starting with some initial game position, the algorithm explores the tree of all legal moves down to the requested depth. Scores associated with leaves of the tree are calculated using an evaluation function. A positive score indicates a good position for player A, while a negative score indicates a poor position for player A, and thus a good position for player B. For each player, the transition from one position to another is either maximized (for player A) or minimized (for player B). Each player tries to select his moves in a manner that will be most profitable for him. In searching for the best play for player A, a search of depth 1 tries to determine the immediate move that maximizes the score of the new position.

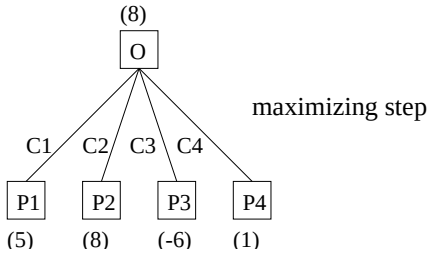


Figure 17.1: Maximizing search at a given location.

In figure 17.1, player A starts at position O, finds four legal moves, constructs these new configurations, and evaluates them. Based on these scores, the best position is P2, with a score of 8. This value is propagated to position O, indicating that this position provides a move to a new position, giving a score of 8 when the player moves to C2. The search of depth 1 is, as a general rule, insufficient, as it does not consider the possible response of an adversary. Such a shallow search results in programs that search greedily for immediate material gains (such as the prize of a queen, in chess) without perceiving that the pieces are protected or that the position is otherwise a losing one (such as a gambit of trading one's queen for a mate). A deeper exploration to depth 2 permits perceiving at least the simplest such countermoves.

Figure 17.2 displays a supplementary analysis of the tree that takes into consideration the possible responses of player B. This search considers B's best moves. For this, the *minimax* algorithm minimizes scores of depth 2.

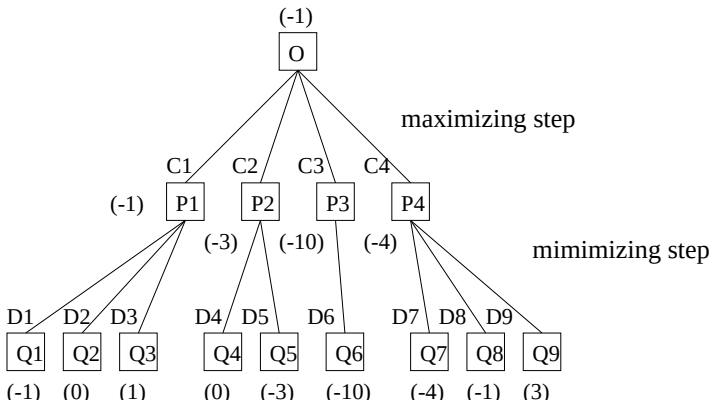


Figure 17.2: Maximizing and minimizing in depth-2 search.

Move P2, which provided an immediate position score of 8, leads to a position with a score of -3. In effect, if B plays D5, then the score of Q5 will be -3. Based on this deeper examination, the move C1 limits the losses with a score of -1, and is thus the preferred move.

In most games, it is possible to try to confuse the adversary, making him play forced moves, trying to muddle the situation in the hope that he will make a mistake. A shallow search of depth 2 would be completely inadequate for this sort of tactic. These sorts of strategies are rarely able to be well exploited by a program because it has no particular vision as to the likely evolution of the positions towards the end of the game.

The difficulty of increased depth of search comes in the form of a combinatorial "explosion." For example, with chess, the exploration of two additional levels adds a factor of around a thousand times more combinations (30×30). Thus, if one searches to a depth of 10, one obtains around 5^{14} positions, which represents too much to search. For this reason, you must try to somehow trim the search tree.

One may note in figure 17.2 that it may be useless to search the branch P3 insofar as the score of this position at depth 1 is poorer than that found in branch P1. In addition the branch P4 does not need to be completely explored. Based on the calculation of Q7, one obtains a score inferior to that of P1, which has already been completely explored. The calculations for Q8 and Q9 cannot improve this situation even if their scores are better than Q7. In a minimizing mode, the poorest score is dropped. The player knows then that these branches provide no useful new options. The minimax variant $\alpha\beta$ uses this approach to decrease the number of branches that must be explored.

Minimax-$\alpha\beta$

We call the α cut the *lower limit* of a maximizing node, and cut β the *upper limit* of a minimizing node. Figure 17.3 shows the cuts carried out in branches P3 and P4 based on knowing the lower limit -1 of P1.

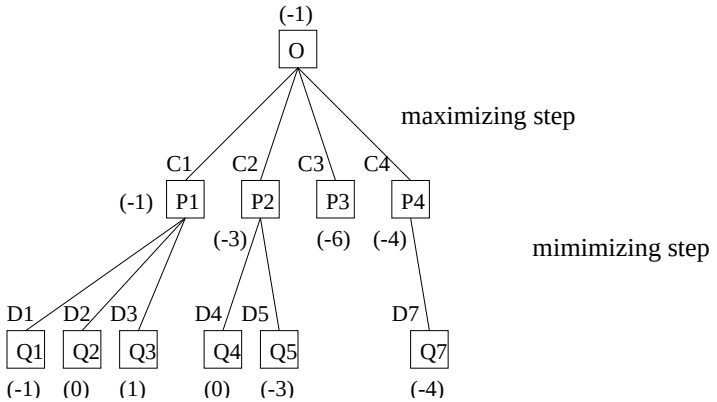


Figure 17.3: Limit α to one level max-min.

As soon as the tree gets broader or deeper the number of cuts increases, thus indicating large subtrees.

A Parametric Module for $\alpha\beta$ *Minimax*

We want to produce a parametric module, `FAlphabeta`, implementing this algorithm, which will be generically reusable for all sorts of two player games. The parameters correspond, on the one hand, to all the information about the proceedings of moves in the game, and on the other hand, to the evaluation function.

Interfaces. We declare two signatures: `REPRESENTATION` to represent plays; and `EVAL` to evaluate a position.

```
# module type REPRESENTATION =
  sig
    type game
    type move
    val game_start : unit → game
    val legal_moves: bool → game → move list
    val play: bool → move → game → game
  end ;;
module type REPRESENTATION =
  sig
    type game
    and move
    val game_start : unit -> game
```

```
    val legal_moves : bool -> game -> move list
    val play : bool -> move -> game -> game
  end

# module type EVAL =
  sig
    type game
    val evaluate: bool → game → int
    val moreI : int
    val lessI: int
    val is_leaf: bool → game → bool
    val is_stable: bool → game → bool
    type state = G | P | N | C
    val state_of : bool → game → state
  end ;;
module type EVAL =
  sig
    type game
    val evaluate : bool -> game -> int
    val moreI : int
    val lessI : int
    val is_leaf : bool -> game -> bool
    val is_stable : bool -> game -> bool
    type state = | G | P | N | C
    val state_of : bool -> game -> state
  end
```

Types `game` and `move` represent abstract types. A player is represented by a boolean value. The function `legal_moves` takes a player and position, and returns the list of possible moves. The function `play` takes a player, a move, and a position, and returns a new position. The values `moreI` and `lessI` are the limits of the values returned by function `evaluate`. The predicate `is_leaf` verifies if a player in a given position can play. The predicate `is_stable` indicates whether the position for the player represents a stable position. The results of these functions influence the pursuit of the exploration of moves when one attains the specified depth.

The signature `ALPHABETA` corresponds to the signature resulting from the complete application of the parametric module that one wishes to use. These hide the different auxiliary functions that we use to implement the algorithm.

```
# module type ALPHABETA = sig
    type game
    type move
    val alphabeta : int → bool → game → move
  end ;;
module type ALPHABETA =
  sig type game and move val alphabeta : int -> bool -> game -> move end
```

The function `alphabeta` takes as parameters the depth of the search, the player, and the game position, returning the next move.

We then define the functional signature `FALPHABETA` which must correspond to that of the implementation of the functor.

```
# module type FALPHABETA = functor (Rep : REPRESENTATION)
    → functor (Eval : EVAL with type game = Rep.game)
      → ALPHABETA with type game = Rep.game
                   and type move = Rep.move ;;
module type FALPHABETA =
  functor(Rep : REPRESENTATION) ->
    functor
      (Eval : sig
                type game = Rep.game
                val evaluate : bool -> game -> int
                val moreI : int
                val lessI : int
                val is_leaf : bool -> game -> bool
                val is_stable : bool -> game -> bool
                type state = | G | P | N | C
                val state_of : bool -> game -> state
              end) ->
      sig
        type game = Rep.game
        and move = Rep.move
        val alphabeta : int -> bool -> game -> move
      end
```

Implementation. The parametric module `FAlphabeta0` makes explicit the partition of the type `game` between the two parameters `Rep` and `Eval`. This module has six functions and two exceptions. The player `true` searches to maximize the score while the player `false` seeks to minimize the score. The function `maxmin_iter` calculates the maximum of the best score for the branches based on a move of player `true` and the pruning parameter α.

The function `maxmin` takes four parameters: `depth`, which indicates the actual calculation depth, `node`, a game position, and α and β, the pruning parameters. If the node is a leaf of the tree or if the maximum depth is reached, the function will return its evaluation of the position. If this is not the case, the function applies `maxmin_iter` to all of the legal moves of player `true`, passing it the search function, diminishing the depth remaining (`minmax`). The latter searches to minimize the score resulting from the response of player `false`.

The movements are implemented using exceptions. If the move β is found in the iteration across the legal moves from the function `maxmin`, then it is returned immediately, the value being propagated using an exception. The functions `minmax_iter` and `minmax` provide the equivalents for the other player. The function `search` determines the move to play based on the best score found in the lists of scores and moves.

The principal function `alphabeta` of this module calculates the legal moves from a given position for the requested player, searches down to the requested depth, and returns the best move.

```
# module FAlphabetaO
      (Rep : REPRESENTATION)  (Eval : EVAL with type game = Rep.game)  =
    struct
      type game = Rep.game
      type move = Rep.move
      exception AlphaMovement of int
      exception BetaMovement of int

      let maxmin_iter node minmax_cur beta alpha cp =
        let alpha_resu =
          max alpha (minmax_cur (Rep.play true cp node) beta alpha)
        in if alpha_resu >= beta then  raise (BetaMovement alpha_resu)
        else alpha_resu

      let minmax_iter node maxmin_cur alpha beta cp =
        let beta_resu =
          min beta (maxmin_cur (Rep.play false cp node) alpha beta)
        in if beta_resu <= alpha then raise (AlphaMovement beta_resu)
        else beta_resu

      let rec maxmin depth node alpha beta =
        if (depth < 1 & Eval.is_stable true node)
            or Eval.is_leaf true node
        then Eval.evaluate true node
        else
          try  let prev = maxmin_iter node (minmax (depth - 1)) beta
          in List.fold_left prev alpha (Rep.legal_moves true node)
          with BetaMovement a → a

      and minmax depth node beta alpha =
        if (depth < 1 & Eval.is_stable false node)
            or Eval.is_leaf false node
        then Eval.evaluate false node
        else
          try let prev = minmax_iter node (maxmin (depth - 1)) alpha
          in List.fold_left prev beta (Rep.legal_moves false node)
          with AlphaMovement b → b

      let rec search a l1 l2 = match (l1,l2) with
        (h1 :: q1, h2 :: q2) → if a = h1 then h2  else search a q1 q2
      | ([], [] ) → failwith  ("AB: "^(string_of_int a)^" not found")
      | (_ ,  _) → failwith  "AB: length differs"

      (* val alphabeta : int -> bool -> Rep.game -> Rep.move *)
      let alphabeta depth player level =
        let alpha = ref Eval.lessI and beta = ref Eval.moreI in
        let l = ref [] in
        let cpl = Rep.legal_moves player level in
```

```
        let eval =
          try
            for i = 0 to (List.length cpl) - 1 do
              if player then
                let b = Rep.play player (List.nth cpl i) level in
                let a = minmax (depth-1) b !beta !alpha
                in l := a :: !l ;
                alpha := max !alpha a ;
                (if !alpha >= !beta then raise (BetaMovement !alpha))
              else
                let a = Rep.play player (List.nth cpl i) level in
                let b = maxmin (depth-1) a !alpha !beta
                in l := b :: !l ;
                beta := min !beta b ;
                (if !beta <= !alpha then raise (AlphaMovement !beta))
            done ;
            if player then !alpha else !beta
          with
            BetaMovement a → a
          | AlphaMovement b → b
        in
        l := List.rev !l ;
        search eval !l cpl
    end ;;
module FAlphabeta0 :
  functor(Rep : REPRESENTATION) ->
    functor
      (Eval : sig
                type game = Rep.game
                val evaluate : bool -> game -> int
                val moreI : int
                val lessI : int
                val is_leaf : bool -> game -> bool
                val is_stable : bool -> game -> bool
                type state = | G | P | N | C
                val state_of : bool -> game -> state
              end) ->
      sig
        type game = Rep.game
        and move = Rep.move
        exception AlphaMovement of int
        exception BetaMovement of int
        val maxmin_iter :
          Rep.game ->
          (Rep.game -> int -> int -> int) -> int -> int -> Rep.move -> int
        val minmax_iter :
          Rep.game ->
          (Rep.game -> int -> int -> int) -> int -> int -> Rep.move -> int
        val maxmin : int -> Eval.game -> int -> int -> int
        val minmax : int -> Eval.game -> int -> int -> int
        val search : int -> int list -> 'a list -> 'a
        val alphabeta : int -> bool -> Rep.game -> Rep.move
```

```
    end
```

We may close module `FAlphabetaO` by associating with it the following signature:

```
# module FAlphabeta = (FAlphabetaO : FALPHABETA) ;;
module FAlphabeta : FALPHABETA
```

This latter module may be used with many different game representations and functions to play different games.

Organization of a Game Program

The organization of a program for a two player game may be separated into a portion specific to the game in question as well as a portion applicable to all sorts of games. For this, we propose using several parametric modules parameterized by specific modules, permitting us to avoid the need to rewrite the common portions each time. Figure 17.4 shows the chosen organization.

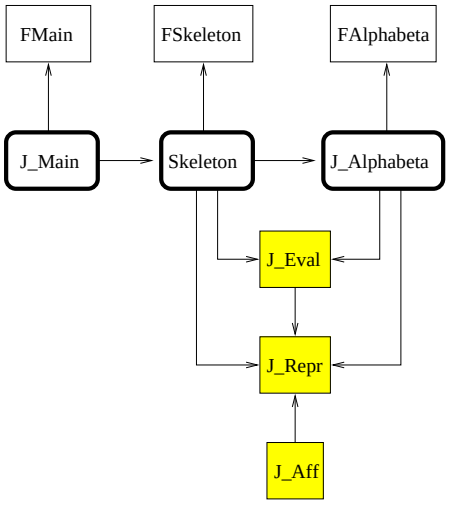


Figure 17.4: Organization of a game application.

The modules with no highlighting correspond to the common parts of the application. These are the parametric modules. We see again the functor `FAlphabeta`. The modules with gray highlighting are the modules designed specifically for a given game. The three principal modules are the representation of the game (`J_Repr`), display of the game (`J_Disp`), and the evaluation function (`J_Eval`). The modules with rounded gray borders are obtained by applying the parametric modules to the simple modules specific to the game.

The module `FAlphabeta` has already been described. The two other common modules are `FMain`, containing the main loop, and `FSkeleton`, that manages the players.

Module `FMain`

Module `FMain` contains the main loop for execution of a game program. It is parameterized using the signature module `SKELETON`, describing the interaction with a player using the following definition:

```
# module type SKELETON = sig
    val home: unit → unit
    val init: unit → ((unit → unit) * (unit → unit))
    val again: unit → bool
    val exit: unit → unit
    val won: unit → unit
    val lost: unit → unit
    val nil: unit → unit
    exception Won
    exception Lost
    exception Nil
  end ;;
module type SKELETON =
  sig
    val home : unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
    val again : unit -> bool
    val exit : unit -> unit
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    exception Won
    exception Lost
    exception Nil
  end
```

The function `init` constructs a pair of action functions for each player. The other functions control the interactions. Module `FMain` contains two functions: `play_game` which alternates between the players, and `main` which controls the main loop.

```
# module FMain (P : SKELETON) =
    struct
      let play_game movements = while true do (fst movements) () ;
        (snd movements) () done

      let main () = let finished = ref false
      in P.home ();
      while not !finished do
        ( try  play_game (P.init ())
        with P.Won → P.won ()
```

```
        | P.Lost  → P.lost ()
        | P.Nil   → P.nil () );
        finished := not (P.again ())
      done ;
      P.exit ()
  end ;;
module FMain :
  functor(P : SKELETON) ->
    sig
      val play_game : (unit -> 'a) * (unit -> 'b) -> unit
      val main : unit -> unit
    end
```

Module **FSkeleton**

Parametric module `FSkeleton` controls the moves of each player according to the rules provided at the start of the section based on the nature of the players (automated or not) and the order of the players. It needs various parameters to represent the game, game states, the evaluation function, and the $\alpha\beta$ search as described in figure 17.4.

We start with the signature needed for game display.

```
# module type DISPLAY = sig
    type game
    type move
    val home: unit → unit
    val exit: unit → unit
    val won: unit → unit
    val lost: unit → unit
    val nil: unit → unit
    val init: unit → unit
    val position : bool → move → game → game → unit
    val choice : bool →  game → move
    val q_player : unit → bool
    val q_begin : unit → bool
    val q_continue : unit → bool
  end ;;
module type DISPLAY =
  sig
    type game
    and move
    val home : unit -> unit
    val exit : unit -> unit
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : bool -> move -> game -> game -> unit
    val choice : bool -> game -> move
    val q_player : unit -> bool
```

```
    val q_begin : unit -> bool
    val q_continue : unit -> bool
  end
```

It is worth noting that the representation of the game and of the moves must be shared by all the parametric modules, which constrain the types. The two principal functions are `playH` and `playM`, respectively controlling the move of a human player (using the function `Disp.choice`) and that of an automated player. The function `init` determines the nature of the players and the sorts of responses for `Disp.q_player`.

```
# module FSkeleton
      (Rep   : REPRESENTATION)
      (Disp   : DISPLAY with type game = Rep.game and  type move = Rep.move)
      (Eval  : EVAL with type game = Rep.game)
      (Alpha : ALPHABETA with type game = Rep.game and  type move = Rep.move) =
    struct
      let depth = ref 4
      exception Won
      exception Lost
      exception Nil
      let won = Disp.won
      let lost = Disp.lost
      let nil = Disp.nil
      let again = Disp.q_continue
      let play_game = ref (Rep.game_start())
      let exit = Disp.exit
      let home = Disp.home

      let playH player () =
        let choice = Disp.choice player !play_game in
        let old_game = !play_game
        in play_game := Rep.play player choice !play_game ;
        Disp.position player choice old_game !play_game ;
        match Eval.state_of player !play_game with
          Eval.P → raise Lost
        | Eval.G → raise Won
        | Eval.N → raise Nil
        | _      → ()

      let playM player () =
        let choice = Alpha.alphabeta !depth player !play_game in
        let old_game = !play_game
        in play_game := Rep.play player choice !play_game ;
        Disp.position player choice old_game !play_game ;
        match Eval.state_of player !play_game with
          Eval.G → raise Won
        | Eval.P → raise Lost
        | Eval.N → raise Nil
        | _      → ()
```

```
      let init () =
        let a = Disp.q_player () in
        let b = Disp.q_player()
        in play_game := Rep.game_start () ;
        Disp.init () ;
        match (a,b) with
          true,true   → playM true, playM false
        | true,false  → playM true, playH false
        | false,true  → playH true, playM false
        | false,false → playH true, playH false
  end ;;
module FSkeleton :
  functor(Rep : REPRESENTATION) ->
    functor
      (Disp : sig
                type game = Rep.game
                and move = Rep.move
                val home : unit -> unit
                val exit : unit -> unit
                val won : unit -> unit
                val lost : unit -> unit
                val nil : unit -> unit
                val init : unit -> unit
                val position : bool -> move -> game -> game -> unit
                val choice : bool -> game -> move
                val q_player : unit -> bool
                val q_begin : unit -> bool
                val q_continue : unit -> bool
              end) ->
      functor
        (Eval : sig
                  type game = Rep.game
                  val evaluate : bool -> game -> int
                  val moreI : int
                  val lessI : int
                  val is_leaf : bool -> game -> bool
                  val is_stable : bool -> game -> bool
                  type state = | G | P | N | C
                  val state_of : bool -> game -> state
                end) ->
          functor
            (Alpha : sig
                       type game = Rep.game
                       and move = Rep.move
                       val alphabeta : int -> bool -> game -> move
                     end) ->
            sig
              val depth : int ref
              exception Won
              exception Lost
              exception Nil
              val won : unit -> unit
```

```
  val lost : unit -> unit
  val nil : unit -> unit
  val again : unit -> bool
  val play_game : Disp.game ref
  val exit : unit -> unit
  val home : unit -> unit
  val playH : bool -> unit -> unit
  val playM : bool -> unit -> unit
  val init : unit -> (unit -> unit) * (unit -> unit)
end
```

The independent parts of the game are thus implemented. One may then begin programming different sorts of games. This modular organization facilitates making modifications to the movement scheme or to the evaluation function for a game as we shall soon see.

Connect Four

We will next examine a simple game, a vertical tic-tac-toe, known as Connect Four. The game is represented by seven columns each consisting of six lines. In turn, a player places on a column a piece of his color, where it then falls down to the lowest free location in this column. If a column is completely filled, neither player is permitted to play there. The game ends when one of the players has built a line of four pieces in a row (horizontal, vertical, or diagonal), at which point this player has won, or when all the columns are filled with pieces, in which the outcome is a draw. Figure 17.5 shows a completed game.

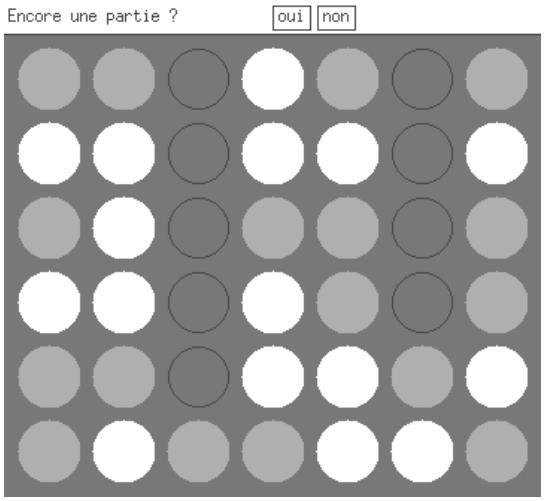


Figure 17.5: An example of Connect Four.

Note the "winning" line of four gray pieces in a diagonal, going down and to the right.

Game Representation: module `C4_rep`. We choose for this game a matrix-based representation. Each element of the matrix is either empty, or contains a player's piece. A move is numbered by the column. The legal moves are the columns in which the final (top) row is not filled.

```
# module C4_rep = struct
    type cell = A | B | Empty
    type game = cell array array
    type move = int
    let col = 7 and row = 6
    let game_start () = Array.create_matrix row col Empty

    let legal_moves b m =
      let l = ref [] in
      for c = 0 to col-1 do if m.(row-1).(c) = Empty then l := (c+1) :: !l done;
      !l

    let augment mat c =
      let l = ref row
      in while !l > 0 & mat.(!l-1).(c-1) = Empty do  decr l done ; !l + 1

    let player_gen cp m e =
      let mj = Array.map Array.copy  m
      in  mj.((augment mj cp)-1).(cp-1) <- e ; mj

    let play b cp m = if b then player_gen cp m A else player_gen cp m B
  end ;;
module C4_rep :
  sig
    type cell = | A | B | Empty
    and game = cell array array
    and move = int
    val col : int
    val row : int
    val game_start : unit -> cell array array
    val legal_moves : 'a -> cell array array -> int list
    val augment : cell array array -> int -> int
    val player_gen : int -> cell array array -> cell -> cell array array
    val play : bool -> int -> cell array array -> cell array array
  end
```

We may easily verify if this module accepts the constraints of the signature `REPRESENTATION`.

```
# module C4_rep_T = (C4_rep : REPRESENTATION) ;;
module C4_rep_T : REPRESENTATION
```

Game Display: Module `C4_text`. Module `C4_text` describes a text-based interface for the game Connect Four that is compatible with the signature `DISPLAY`. It it is not particularly sophisticated, but, nonetheless, demonstrates how modules are assembled together.

```
# module C4_text = struct
    open C4_rep
    type game = C4_rep.game
    type move = C4_rep.move

    let print_game mat =
      for l = row - 1 downto 0 do
        for c = 0 to col - 1 do
          match mat.(l).(c) with
            A     → print_string "X "
          | B     → print_string "O "
          | Empty → print_string ". "
        done;
        print_newline ()
      done ;
      print_newline ()

    let home () = print_string "C4 ...\n"
    let exit () = print_string "Bye for now ... \n"
    let question s =
      print_string s;
      print_string " y/n ? " ;
      read_line() = "y"
    let q_begin () = question "Would you like to begin?"
    let q_continue () = question "Play again?"
    let q_player () = question "Is there to be a machine player ?"

    let won  ()= print_string "The first player won" ; print_newline ()
    let lost () = print_string "The first player lost" ; print_newline ()
    let nil () = print_string "Stalemate" ; print_newline ()

    let init () =
      print_string "X: 1st player   O: 2nd player";
      print_newline () ; print_newline () ;
      print_game (game_start ()) ; print_newline()

    let position b c aj j = print_game j

    let is_move = function '1'..'7' → true | _ → false

    exception Move of int
    let rec choice player game =
      print_string ("Choose player" ^ (if player then "1" else "2") ^ " : ") ;
      let l = legal_moves player game
      in try  while true do
        let i = read_line()
```

```
        in ( if (String.length i > 0)  && (is_move  i.[0])
        then let c = (int_of_char i.[0]) - (int_of_char '0')
        in if List.mem c l then raise (Move c) );
        print_string "Invalid move - try again"
      done ;
        List.hd l
      with  Move i → i
      | _ → List.hd l
  end ;;
module C4_text :
  sig
    type game = C4_rep.game
    and move = C4_rep.move
    val print_game : C4_rep.cell array array -> unit
    val home : unit -> unit
    val exit : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val q_player : unit -> bool
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : 'a -> 'b -> 'c -> C4_rep.cell array array -> unit
    val is_move : char -> bool
    exception Move of int
    val choice : bool -> C4_rep.cell array array -> int
  end
```

We may immediately verify that this conforms to the constraints of the signature DISPLAY

```
# module C4_text_T = (C4_text : DISPLAY) ;;
module C4_text_T : DISPLAY
```

Evaluation Function: module C4_eval. The quality of a game player depends primarily on the position evaluation function. Module C4_eval defines evaluate, which evaluates the value of a position for the specified player. This function calls eval_bloc for the four compass directions as well as the diagonals. eval_bloc then calls eval_four to calculate the number of pieces in the requested line. Table value provides the value of a block containing 0, 1, 2, or 3 pieces of the same color. The exception Four is raised when 4 pieces are aligned.

```
# module C4_eval = struct open C4_rep type game = C4_rep.game
    let value =
      Array.of_list [0; 2; 10; 50]
    exception Four of int
```

```
exception Nil_Value
exception Arg_invalid
let lessI = -10000
let moreI = 10000
let eval_four m l_dep c_dep delta_l delta_c =
  let n = ref 0 and e = ref Empty
  and x = ref c_dep  and  y = ref l_dep
  in try
    for i = 1 to 4 do
      if !y<0 or !y>=row or !x<0 or !x>=col then raise Arg_invalid ;
      ( match m.(!y).(!x) with
        A    → if !e = B then raise Nil_Value ;
          incr n ;
          if !n = 4 then raise (Four moreI) ;
          e := A
      | B    → if !e = A then raise Nil_Value ;
          incr n ;
          if !n = 4 then raise (Four lessI);
          e := B;
      | Empty → () ) ;
          x := !x + delta_c ;
      y := !y + delta_l
    done ;
    value.(!n) * (if !e=A then 1 else -1)
  with
    Nil_Value | Arg_invalid  → 0

let eval_bloc m e cmin cmax lmin lmax dx dy =
  for c=cmin to cmax do for l=lmin to lmax do
    e := !e + eval_four m l c dx dy
  done done

let evaluate b m =
  try let evaluation = ref 0
  in (* evaluation of rows *)
  eval_bloc m evaluation 0 (row-1) 0 (col-4) 0 1 ;
  (* evaluation of columns *)
  eval_bloc m evaluation 0 (col-1) 0 (row-4) 1 0 ;
  (* diagonals coming from the first line (to the right) *)
  eval_bloc m evaluation 0 (col-4) 0 (row-4) 1 1 ;
  (* diagonals coming from the first line (to the left) *)
  eval_bloc m evaluation 1 (row-4) 0 (col-4) 1 1 ;
  (* diagonals coming from the last line (to the right) *)
  eval_bloc m evaluation 3 (col-1) 0 (row-4) 1 (-1) ;
  (* diagonals coming from the last line (to the left) *)
  eval_bloc m evaluation 1 (row-4) 3 (col-1) 1 (-1) ;
  !evaluation
  with Four v → v

let is_leaf b m = let v = evaluate b m
in v=moreI or v=lessI or legal_moves b m = []
```

```
    let is_stable b j = true

    type state = G | P | N | C

    let state_of player m =
      let v = evaluate player m
      in if v = moreI then if player then G else P
      else if v = lessI then if player then P else G
      else if legal_moves player m = [] then N else C
  end ;;
module C4_eval :
  sig
    type game = C4_rep.game
    val value : int array
    exception Four of int
    exception Nil_Value
    exception Arg_invalid
    val lessI : int
    val moreI : int
    val eval_four :
      C4_rep.cell array array -> int -> int -> int -> int -> int
    val eval_bloc :
      C4_rep.cell array array ->
      int ref -> int -> int -> int -> int -> int -> int -> unit
    val evaluate : 'a -> C4_rep.cell array array -> int
    val is_leaf : 'a -> C4_rep.cell array array -> bool
    val is_stable : 'a -> 'b -> bool
    type state = | G | P | N | C
    val state_of : bool -> C4_rep.cell array array -> state
  end
```

Module `C4_eval` is compatible with the constraints of signature `EVAL`.

```
# module C4_eval_T = (C4_eval : EVAL) ;;
module C4_eval_T : EVAL
```

To play two evaluation functions against one another, it is necessary to modify `evaluate` to apply the proper evaluation function for each player.

Assembly of the modules All the components needed to realize the game of Connect Four are now implemented. We only need assemble them together based on the schema of diagram 17.4. First, we construct `C4_skeleton`, which is the application of parameter module `FSkeleton` to modules `C4_rep`, `C4_text`, `C4_eval` and the result of the application of parametric module `FAlphaBeta` to `C4_rep` and `C4_eval`.

```
# module C4_skeleton =
```

```
    FSkeleton (C4_rep) (C4_text) (C4_eval)  (FAlphabeta (C4_rep) (C4_eval)) ;;
module C4_skeleton :
  sig
    val depth : int ref
    exception Won
    exception Lost
    exception Nil
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val again : unit -> bool
    val play_game : C4_text.game ref
    val exit : unit -> unit
    val home : unit -> unit
    val playH : bool -> unit -> unit
    val playM : bool -> unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
  end
```

We then obtain the principal module `C4_main` by applying parametric module `FMain` on the result of the preceding application `C4_skeleton`

```
# module C4_main = FMain(C4_skeleton) ;;
module C4_main :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

The game is initiated by the application of function `C4_main.main` on `()`.

Testing the Game. Once the general game skeleton has been written, games may be played in various ways. Two human players may play against each other, with the program merely verifying the validity of the moves; a person may play against a programmed player; or programs may play against each other. While this last mode might not be interesting for the human, it does make it easy to run tests without having to wait for a person's responses. The following game demonstrates this scenario.

```
# C4_main.main () ;;
C4 ...
Is there to be a machine player ? y/n ? y
Is there to be a machine player ? y/n ? y
X: 1st player   O: 2nd player

. . . . . .
. . . . . .
. . . . . .
```

```
. . . . . .
. . . . . .
. . . . . .
```

Once the initial position is played, player 1 (controlled by the program) calculates its move which is then applied.

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . X .
```

Player 2 (always controlled by the program) calculates its response and the game proceeds, until a game-ending move is found. In this example, player 1 wins the game based on the following final position:

```
. O O O . O .
. X X X . X .
X O O X . O .
X X X O . X .
X O O X X O .
X O O O X X O
Player 1 wins
Play again(y/n) ? n
Good-bye ...
- : unit = ()
```

Graphical Interface. To improve the enjoyment of the game, we define a graphical interface for the program, by defining a new module, `C4_graph`, compatible with the signature `DISPLAY`, which opens a graphical window, controlled by mouse clicks. The text of this module may be found in the subdirectory `Applications` on the CD-ROM (see page 1).

```
# module C4_graph = struct
    open C4_rep
    type game = C4_rep.game
    type move = C4_rep.move
    let r = 20          (* color of piece *)
    let ec = 10         (* distance between pieces *)
    let dec = 30        (* center of first piece *)
    let cote = 2*r + ec (* height of a piece looked at like a checker *)
    let htexte = 25     (* where to place text *)
    let width = col * cote + ec          (* width of the window *)
    let height = row * cote + ec + htexte (* height of the window *)
```

```
  let height_of_game = row * cote + ec   (* height of game space *)
  let hec = height_of_game + 7  (* line for messages *)
  let lec = 3                    (* columns for messages *)
  let margin = 4                  (* margin for buttons *)
  let xb1 = width / 2   (* position x of button1 *)
  let xb2 = xb1 + 30      (* position x of button2 *)
  let yb = hec - margin    (* position y of the buttons *)
  let wb = 25             (* width of the buttons *)
  let hb = 16             (* height of the buttons *)

(* val t2e : int -> int *)
(* Convert a matrix coordinate into a graphical coordinate *)
  let t2e i = dec + (i-1)*cote

(* The Colors *)
  let cN = Graphics.black   (* trace *)
  let cA = Graphics.red     (* Human player *)
  let cB = Graphics.yellow  (* Machine player *)
  let cF = Graphics.blue    (* Game Background color *)
(* val draw_table : unit -> unit :  Trace an empty table *)
  let draw_table () =
    Graphics.clear_graph();
    Graphics.set_color cF;
    Graphics.fill_rect 0 0 width height_of_game;
    Graphics.set_color cN;
    Graphics.moveto 0 height_of_game;
    Graphics.lineto width height_of_game;
    for l = 1 to row do
      for c = 1 to col do
        Graphics.draw_circle (t2e c) (t2e l) r
      done
    done

(* val draw_piece : int -> int -> Graphics.color -> unit *)
(* 'draw_piece l c co' draws a piece of color co at coordinates l c *)
  let draw_piece l c col =
    Graphics.set_color col;
    Graphics.fill_circle (t2e c) (t2e l) (r+1)

(* val augment : Rep.item array array -> int -> Rep.move *)
(* 'augment m c' redoes the line or drops the piece for c in m *)
  let augment mat c =
    let l = ref row in
    while !l > 0 & mat.(!l-1).(c-1) = Empty do
      decr l
    done;
    !l

(* val conv : Graphics.status -> int *)
(* convert the region where player has clicked in controlling the game *)
  let conv st =
```

```
    (st.Graphics.mouse_x - 5) / 50 + 1

(* val wait_click : unit -> Graphics.status *)
(* wait for a mouse click *)
  let wait_click () = Graphics.wait_next_event [Graphics.Button_down]

(* val choiceH : Rep.game -> Rep.move *)
(* give opportunity to the human player to choose a move *)
(* the function offers possible moves *)
  let rec choice player  game  =
    let c = ref 0 in
    while not ( List.mem !c (legal_moves player game) ) do
      c := conv ( wait_click() )
    done;
    !c
(* val home : unit -> unit : home screen *)
  let home () =
    Graphics.open_graph
      (" " ^ (string_of_int width) ^ "x" ^ (string_of_int height) ^ "+50+50");
    Graphics.moveto (height/2) (width/2);
    Graphics.set_color cF;
    Graphics.draw_string "C4";
    Graphics.set_color cN;
    Graphics.moveto 2 2;
    Graphics.draw_string "by Romuald COEFFIER & Mathieu DESPIERRE";
    ignore (wait_click ());
    Graphics.clear_graph()

(* val end : unit -> unit ,  the end of the game *)
  let exit () = Graphics.close_graph()

(* val draw_button : int -> int -> int -> int -> string -> unit *)
(* 'draw_button x y w h s' draws a rectangular button at coordinates *)
(* x,y with width w and height h and appearance s *)
  let draw_button x y w h s =
    Graphics.set_color cN;
    Graphics.moveto x y;
    Graphics.lineto x (y+h);
    Graphics.lineto (x+w) (y+h);
    Graphics.lineto (x+w) y;
    Graphics.lineto x y;
    Graphics.moveto (x+margin) (hec);
    Graphics.draw_string s

(* val draw_message : string -> unit  * position message s *)
  let draw_message s =
    Graphics.set_color cN;
    Graphics.moveto lec hec;
    Graphics.draw_string s

(* val erase_message : unit -> unit  erase the starting position *)
```

```
  let erase_message () =
    Graphics.set_color Graphics.white;
    Graphics.fill_rect 0 (height_of_game+1) width htexte

(* val question : string -> bool *)
(* 'question s' poses the question s, the response being obtained by *)
(*    selecting one of two buttons, 'yes' (=true) and 'no' (=false) *)
  let question s =
    let rec attente () =
      let e = wait_click () in
      if (e.Graphics.mouse_y < (yb+hb)) & (e.Graphics.mouse_y > yb) then
        if (e.Graphics.mouse_x > xb1) & (e.Graphics.mouse_x < (xb1+wb)) then
          true
        else
          if (e.Graphics.mouse_x > xb2) & (e.Graphics.mouse_x < (xb2+wb)) then
            false
          else
            attente()
      else
      attente () in
    draw_message s;
    draw_button xb1 yb wb hb "yes";
    draw_button xb2 yb wb hb "no";
    attente()

(* val q_begin : unit -> bool *)
(* Ask, using function 'question', if the player wishes to start *)
(* (yes=true) *)
  let q_begin () =
    let b = question "Would you like to begin ?" in
    erase_message();
    b

(* val q_continue : unit -> bool *)
(* Ask, using function 'question', if the player wishes to play again *)
(* (yes=true) *)
  let q_continue () =
    let b = question "Play again ?" in
    erase_message();
    b

  let q_player () =
    let b = question "Is there to be a machine player?" in
    erase_message ();
    b
(* val won : unit -> unit *)
(* val lost : unit -> unit *)
(* val nil : unit -> unit *)
(* Three functions for these three cases *)
  let won () =
    draw_message "I won :-)" ; ignore (wait_click ()) ; erase_message()
```

```
    let lost () =
      draw_message "You won :-("; ignore (wait_click ()) ; erase_message()
    let nil () =
      draw_message "Stalemate" ; ignore (wait_click ()) ; erase_message()

  (* val init : unit -> unit *)
  (* This is called at every start of the game for the position *)
    let init  = draw_table

    let position b c aj nj  =
      if b then
        draw_piece (augment nj c) c cA
      else
        draw_piece (augment nj c) c cB

  (* val drawH : int -> Rep.item array array -> unit *)
  (* Position when the human player chooses move cp in situation j *)
    let drawH cp j =  draw_piece (augment j cp) cp cA

  (* val drawM : int -> cell array array -> unit*)
  (* Position when the machine player chooses move cp in situation j *)
    let drawM cp j =  draw_piece (augment j cp) cp cB
  end ;;
module C4_graph :
  sig
    type game = C4_rep.game
    and move = C4_rep.move
    val r : int
    val ec : int
    val dec : int
    val cote : int
    val htexte : int
    val width : int
    val height : int
    val height_of_game : int
    val hec : int
    val lec : int
    val margin : int
    val xb1 : int
    val xb2 : int
    val yb : int
    val wb : int
    val hb : int
    val t2e : int -> int
    val cN : Graphics.color
    val cA : Graphics.color
    val cB : Graphics.color
    val cF : Graphics.color
    val draw_table : unit -> unit
    val draw_piece : int -> int -> Graphics.color -> unit
    val augment : C4_rep.cell array array -> int -> int
    val conv : Graphics.status -> int
```

```
    val wait_click : unit -> Graphics.status
    val choice : 'a -> C4_rep.cell array array -> int
    val home : unit -> unit
    val exit : unit -> unit
    val draw_button : int -> int -> int -> int -> string -> unit
    val draw_message : string -> unit
    val erase_message : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val q_player : unit -> bool
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : bool -> int -> 'a -> C4_rep.cell array array -> unit
    val drawH : int -> C4_rep.cell array array -> unit
    val drawM : int -> C4_rep.cell array array -> unit
  end
```

We may also create a new skeleton (`C4_skeletonG`) which results from the application of parametric module `FSkeleton`.

```
# module C4_skeletonG =
    FSkeleton (C4_rep) (C4_graph) (C4_eval) (FAlphabeta (C4_rep) (C4_eval)) ;;
```

Only the display parameter differs from the text version application of `FSkeleton`. We may thereby create a principal module for Connect Four with a graphical user interface.

```
# module C4_mainG = FMain(C4_skeletonG) ;;
module C4_mainG :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

The evaluation of the expression `C4_mainG.main()` opens a graphical window as in figure 17.5 and controls the interaction with the user.

Stonehenge

Stonehenge, created by Reiner Knizia, is a game involving construction of "ley-lines." The rules are simple to understand but our interest in the game lies in its high number of possible moves. The rules may be found at:

Link: http://www.cix.co.uk/~convivium/files/stonehen.htm

The initial game position is represented in figure 17.6.

Game Presentation

The purpose of the game is to win at least 8 "ley-lines" (clear lines) out of the 15 available. One gains a line by positioning pieces (or megaliths) on gray positions along a ley-line.

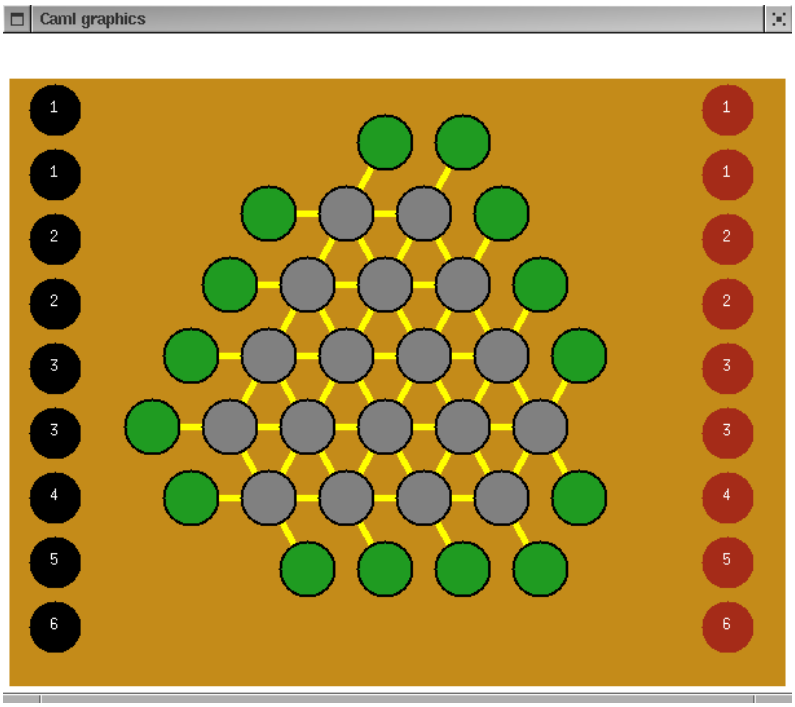


Figure 17.6: Initial position of Stonehenge.

In turn, each player places one of his 9 pieces, numbered from 1 to 6, on one of the 18 gray internal positions. They may not place a piece on a position that is already occupied. Each time a piece is placed, one or several ley-lines may be won or lost.

A ley-line is won by a player if the total of the values of his pieces on the line is greater than the total of the pieces for the other player. There may be empty spaces left if the opponent has no pieces left that would allow winning the line.

For example in figure 17.7, the black player starts by placing the piece of value 3, the red player his "2" piece, then the black player plays the "6" piece, winning a line.

Red then plays the "4" piece, also winning a ley-line. This line has not been completely filled, but red has won because there is no way for black to overcome red's score.

Note that the red player might just as well have played "3" rather than "4," and still won the line. In effect, there is only one free case for this ley-line where the strongest black piece has a value of 5, and so black cannot beat red for this particular line.

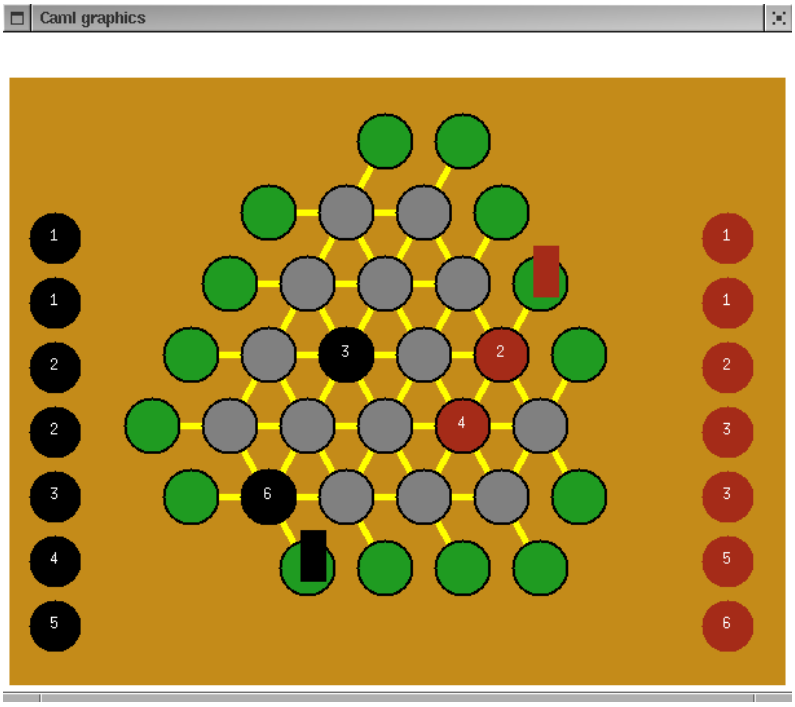


Figure 17.7: Position after 4 moves.

In the case where the scores are equal across a full line, whoever placed the last piece *without* having beaten his adversary's score, loses the line. Figure 17.8 demonstrates such a situation.

The last red move is piece "4". On the full line where the "4" is placed, the scores are equal. Since red was the last player to have placed a piece, but did not beat his adversary, red loses the line, as indicated by a black block.

We may observe that the function `play` fills the role of arbitrating and accounting for these subtleties in the placement of lines.

There can never be a tie in this game. There are 15 lines, each of which will be accounted for at some point in the game, at which point one of the players will have won at least 8 lines.

Search Complexity

Before completely implementing a new game, it is important to estimate the number of legal moves between two moves in a game, as well as the number of possible moves

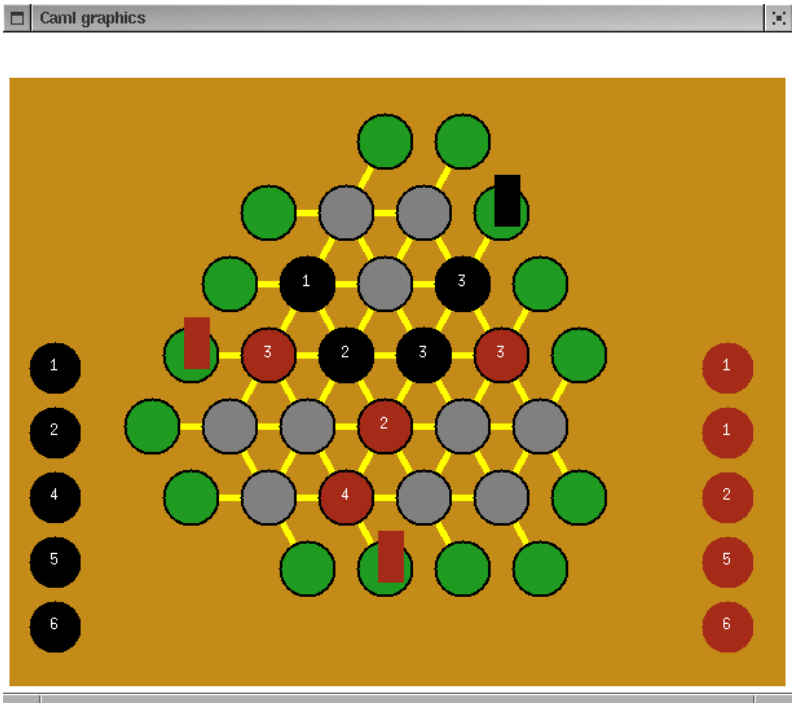


Figure 17.8: Position after 6 moves.

for each side. These values may be used to estimate a reasonable maximum depth for the minimax-$\alpha\beta$ algorithm.

In the game Stonehenge, the number of moves for each side is initially based on the number of pieces for the two players, that is, 18. The number of possible moves diminishes as the game progresses. At the first move, the player has 6 different pieces and 18 positions free. At the second move, the second player has 6 different pieces, and 17 positions in which they may be placed (102 legal moves). Moving from a depth of 2 to 4 for the initial moves of the game results in the number of choices going from about 10^4 to about 10^8.

On the other hand, near the end of the game, in the final 8 moves, the complexity is greatly reduced. If we take a pessimistic calculation (where all pieces are different), we obtain about 23 million possibilities:

$$4 * 8 * 4 * 7 * 3 * 6 * 3 * 5 * 2 * 4 * 2 * 3 * 1 * 2 * 1 * 1 = 23224320$$

It might seem appropriate to calculate with a depth of around 2 for the initial set of moves. This may depend on the evaluation function, and on its ability to evaluate the positions at the start of the game, when there are few pieces in place. On the other

hand, near the end of the game, the depth may readily be increased to around 4 or 6, but this would probably be too late a point to recover from a weak position.

Implementation

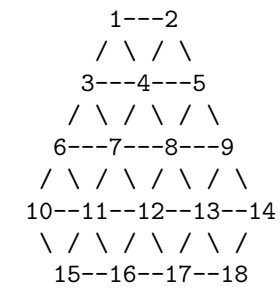
We jump straight into describing the game representation and arbitration so that we may concentrate on the evaluation function.

The implementation of this game follows the architecture used for Connect Four, described in figure 17.4. The two principal difficulties will be to follow the game rules for the placement of pieces, and the evaluation function, which must be able to evaluate positions as quickly as possible while remaining useful.

Game Representation. There are four notable data structures in this game:

- the pieces of the players (type *piece*),
- the positions (type *placement*),
- the 15 ley-lines,
- the 18 locations where pieces may be placed.

We provide a unique number for each location:



Each location participates in 3 ley-lines. We also number each ley-line. This description may be found in the declaration of the list *lines*, which is converted to a vector (`vector_l`). A location is either empty, or contains a piece that has been placed, and the piece's possessor. We also store, for each location, the number of the lines that pass through it. This table is calculated by `lines_per_case` and is named `num_line_per_case`.

The game is represented by the vector of 18 cases, the vector of 15 ley-lines either won or not, and the lists of pieces left for the two players. The function `game_start` creates these four elements.

The calculation of a player's legal moves resolves into a Cartesian product of the pieces available against the free positions. Various utility functions allow counting the score of a player on a line, calculating the number of empty locations on a line, and verifying

if a line has already been won. We only need to implement `play` which plays a move and decides which pieces to place. We write this function at the end of the listing of module `Stone_rep`.

```
# module Stone_rep = struct
    type player = bool
    type piece =  P of int
    let int_of_piece = function P x → x
    type placement = None | M of player
    type case = Empty | Occup of player*piece
    let value_on_case = function
        Empty → 0
      | Occup (j, x) → int_of_piece x

    type game = J of case array * placement array * piece list * piece list
    type move =  int * piece

    let lines = [
      [0;1]; [2;3;4]; [5; 6; 7; 8;]; [9; 10; 11; 12; 13]; [14; 15; 16; 17];
      [0; 2; 5; 9]; [1; 3; 6; 10; 14]; [4; 7; 11; 15]; [8; 12; 16]; [13; 17];
      [9; 14]; [5; 10; 15]; [2; 6; 11; 16]; [0; 3; 7; 12; 17]; [1; 4; 8; 13]  ]

    let vector_l = Array.of_list lines

    let lines_per_case v =
      let t =  Array.length v  in
      let r = Array.create 18 [||] in
      for i = 0 to 17  do
        let w = Array.create 3 0
        and p = ref 0 in
        for j=0 to t-1 do if List.mem i v.(j) then (w.(!p) <- j; incr p)
        done;
        r.(i) <- w
      done;
      r

    let num_line_per_case = lines_per_case vector_l
    let rec lines_of_i i l = List.filter (fun t → List.mem i t) l

    let lines_of_cases l =
      let a = Array.create 18 l in
      for i=0 to 17 do
        a.(i) <-  (lines_of_i i l)
      done; a

    let ldc = lines_of_cases lines

    let game_start ()= let lp = [6; 5; 4; 3; 3; 2; 2; 1; 1] in
    J ( Array.create 18 Empty, Array.create 15 None,
        List.map (fun x → P x) lp, List.map (fun x → P x) lp )

    let rec unicity l = match l with
```

```
   [] → []
| h :: t → if List.mem h t then unicity t else h :: (unicity t)

let legal_moves player (J (ca, m, r1, r2)) =
  let r = if player then r1 else r2 in
  if r = [] then []
  else
    let l = ref [] in
    for i = 0 to 17 do
      if value_on_case ca.(i) = 0 then l:= i :: !l
    done;
    let l2 = List.map (fun x→
      List.map (fun y→ x,y) (List.rev(unicity r)) ) !l in
    List.flatten l2
let copy_board p = Array.copy p

let  carn_copy m = Array.copy m
let rec play_piece stone l = match l with
   [] → []
| x :: q → if x=stone  then q
else x :: (play_piece stone q)

let count_case player case = match case with
  Empty → 0
| Occup (j,p) → if j = player then (int_of_piece p) else  0

let count_line player line pos =
  List.fold_left (fun x y → x +  count_case player pos.(y)) 0 line

let rec count_max n = function
    [] → 0
  | t :: q →
      if (n>0) then
        (int_of_piece t) + count_max (n-1) q
      else
        0

let rec nbr_cases_free ca  l  = match l with
   []   → 0
| t :: q → let c = ca.(t) in
  match c with
    Empty → 1 + nbr_cases_free ca q
  | _     → nbr_cases_free ca q

let a_placement i ma =
  match ma.(i) with
    None → false
  | _ → true

let which_placement i ma =
  match ma.(i) with
```

```
        None → failwith "which_placement"
      | M j  → j

    let is_filled l ca =  nbr_cases_free ca l = 0

    (* function play : arbitrates the game *)
    let play player move game =
      let (c, i) = move in
      let J (p, m, r1, r2) = game in
      let nr1,nr2 = if player then play_piece i r1,r2
      else r1,  play_piece i r2  in
      let np = copy_board p in
      let nm = carn_copy m in
      np.(c)<-Occup(player,i);  (* on play le move *)
      let lines_of_the_case = num_line_per_case.(c) in

      (* calculation of the placements of the three lines *)
      for k=0 to 2 do
        let l =  lines_of_the_case.(k) in
        if not (a_placement l nm) then (
          if is_filled vector_l.(l) np then (
            let c1 = count_line player vector_l.(l) np
            and c2 = count_line (not player) vector_l.(l) np in
            if (c1 > c2) then nm.(l) <- M player
            else  ( if c2 > c1 then nm.(l) <- M (not player)
            else nm.(l) <- M (not player ))))
      done;

      (* calculation of other placements *)
      for k=0 to 14 do
        if not (a_placement k nm ) then
          if is_filled vector_l.(k) np then failwith "player"
          else
            let c1 = count_line player  vector_l.(k) np
            and c2 = count_line (not player) vector_l.(k) np in
            let cases_free = nbr_cases_free np vector_l.(k) in
            let max1  = count_max cases_free
                (if player then nr1 else nr2)
            and max2 = count_max cases_free
                (if player then nr2 else nr1) in
            if c1 >= c2 + max2 then nm.(k) <- M player
            else if c2 >= c1 + max1 then nm.(k) <- M (not player)
      done;
      J(np,nm,nr1,nr2)
   end ;;
module Stone_rep :
  sig
    type player = bool
    and piece = | P of int
    val int_of_piece : piece -> int
    type placement = | None | M of player
    and case = | Empty | Occup of player * piece
```

```
  val value_on_case : case -> int
  type game = | J of case array * placement array * piece list * piece list
  and move = int * piece
  val lines : int list list
  val vector_l : int list array
  val lines_per_case : int list array -> int array array
  val num_line_per_case : int array array
  val lines_of_i : 'a -> 'a list list -> 'a list list
  val lines_of_cases : int list list -> int list list array
  val ldc : int list list array
  val game_start : unit -> game
  val unicity : 'a list -> 'a list
  val legal_moves : bool -> game -> (int * piece) list
  val copy_board : 'a array -> 'a array
  val carn_copy : 'a array -> 'a array
  val play_piece : 'a -> 'a list -> 'a list
  val count_case : player -> case -> int
  val count_line : player -> int list -> case array -> int
  val count_max : int -> piece list -> int
  val nbr_cases_free : case array -> int list -> int
  val a_placement : int -> placement array -> bool
  val which_placement : int -> placement array -> player
  val is_filled : int list -> case array -> bool
  val play : player -> int * piece -> game -> game
end
```

The function `play` decomposes into three stages:

1. Copying the game position and placing a move onto this position;
2. Determination of the placement of a piece on one of the three lines of the case played;
3. Treatment of the other ley-lines.

The second stage verifies that, of the three lines passing through the position of the move, none has already been won, and then checks if they are able to be won. In the latter case, it counts scores for each player and determines which strictly has the greatest score, and attributes the line to the appropriate player. In case of equality, the line goes to the most recent player's adversary. In effect, there are no lines with just one case. A filled line has at least two pieces. Thus if the player which just played has just matched the score of his adversary, he cannot expect to win the line which then goes to his adversary. If the line is not filled, it will be analyzed by "stage 3."

The third stage verifies for each line not yet attributed that it is not filled, and then checks if a player cannot be beaten by his opponent. In this case, the line is immediately given to the opponent. To perform this test, it is necessary to calculate the maximum total potential score of a player on the line (that is, by using his best pieces). If the line is still under dispute, nothing more is done.

Evaluation. The evaluation function must remain simple due to the large number of cases to deal with near the beginning of the game. The idea is not to excessively simplify the game by immediately playing the strongest pieces which would then leave the remainder of the game open for the adversary to play his strong pieces.

We will use two criteria: the number of lines won and an estimate of the potential of future moves by calculating the value of the remaining pieces. We may use the following formula for player 1:

$$score \;=\; 50 * (c_1 - c_2) + 10 * (pr_1 - pr_2)$$

where c_i is the number of lines won, and pr_i is the sum of the pieces remaining for player i.

The formula returns a positive result if the differences between won lines $(c_1 - c_2)$ and the potentials $(pr_1 - pr_2)$ turn to the advantage of player 1. We may see thus that a placement of piece 6 is not appropriate unless it provides a win of at least 2 lines. The gain of one line provides 50, while using the "6" piece costs 10×6 points, so we would thus prefer to play "1" which results in the same score, namely a loss of 10 points.

```
# module Stone_eval = struct
    open Stone_rep
    type game = Stone_rep.game

    exception Done of bool
    let moreI = 1000 and  lessI = -1000

    let nbr_lines_won  (J(ca, m,r1,r2))  =
      let c1,c2 = ref 0,ref 0 in
      for i=0 to 14 do
        if a_placement i m then if which_placement i m then incr c1 else incr c2
      done;
      !c1,!c2

    let rec nbr_points_remaining lig = match lig with
      []  → 0
    | t :: q → (int_of_piece t) + nbr_points_remaining q

    let evaluate player game  =
      let (J (ca,ma,r1,r2)) = game in
      let c1,c2  = nbr_lines_won  game in
      let pr1,pr2 = nbr_points_remaining r1, nbr_points_remaining r2 in
      match player with
        true → if c1 > 7 then moreI else 50 * (c1 - c2) + 10 * (pr1 - pr2)
      | false → if c2 > 7 then lessI else 50 * (c1 - c2) + 10 * (pr1 - pr2)

    let is_leaf player game =
      let v = evaluate player game  in
      v = moreI or v = lessI or legal_moves player game  = []
```

```
    let is_stable player game = true

    type state = G | P | N | C
    let state_of player m =
      let v = evaluate player m in
      if v = moreI then if player then G else P
      else
        if v = lessI
        then if player then P else G
        else
          if legal_moves player  m = [] then  N else C
  end;;
module Stone_eval :
  sig
    type game = Stone_rep.game
    exception Done of bool
    val moreI : int
    val lessI : int
    val nbr_lines_won : Stone_rep.game -> int * int
    val nbr_points_remaining : Stone_rep.piece list -> int
    val evaluate : bool -> Stone_rep.game -> int
    val is_leaf : bool -> Stone_rep.game -> bool
    val is_stable : 'a -> 'b -> bool
    type state = | G | P | N | C
    val state_of : bool -> Stone_rep.game -> state
  end

# module Stone_graph = struct
    open Stone_rep
    type piece = Stone_rep.piece
    type placement = Stone_rep.placement
    type case = Stone_rep.case
    type game = Stone_rep.game
    type move = Stone_rep.move

  (* brightness for a piece *)
    let brightness = 20

  (* the colors  *)
    let cBlack  = Graphics.black
    let cRed    = Graphics.rgb 165 43 24
    let cYellow = Graphics.yellow
    let cGreen  = Graphics.rgb 31 155 33 (*Graphics.green*)
    let cWhite  = Graphics.white
    let cGray   = Graphics.rgb 128 128 128
    let cBlue   = Graphics.rgb 196 139 25 (*Graphics.blue*)

  (* width and height *)
    let width = 600
    let height = 500
  (* the border at the top of the screen from which drawing begins *)
    let top_offset = 30
```

```
(* height of foundaries *)
  let bounds = 5

(* the size of the border on the left side of the virtual table *)
  let virtual_table_xoffset = 145

(* left shift for the black pieces *)
  let choice_black_offset = 40

(* left shift for the red pieces *)
  let choice_red_offset = 560

(* height of a case for the virtual table *)
  let virtual_case_size = 60

(* corresp : int*int -> int*int  *)
(* establishes a correspondence between a location in the matrix *)
(* and a position on the virtual table servant for drawing *)
  let corresp cp =
    match cp with
      0  → (4,1)
    | 1 → (6,1)
    | 2 → (3,2)
    | 3 → (5,2)
    | 4 → (7,2)
    | 5 → (2,3)
    | 6 → (4,3)
    | 7 → (6,3)
    | 8 → (8,3)
    | 9 → (1,4)
    | 10 → (3,4)
    | 11 → (5,4)
    | 12 → (7,4)
    | 13 → (9,4)
    | 14 → (2,5)
    | 15 → (4,5)
    | 16 → (6,5)
    | 17 → (8,5)
    | _ → (0,0)

  let corresp2 ((x,y) as cp) =
    match cp with
      (0,0) → 0
    |  (0,1) → 1
    |  (1,0) → 2
    |  (1,1) → 3
    |  (1,2) → 4
    |  (2,0) → 5
    |  (2,1) → 6
    |  (2,2) → 7
```

```
    |  (2,3) → 8
    |  (3,0) → 9
    |  (3,1) → 10
    |  (3,2) → 11
    |  (3,3) → 12
    |  (3,4) → 13
    |  (4,0) → 14
    |  (4,1) → 15
    |  (4,2) → 16
    |  (4,3) → 17
    |  (x,y) → print_string "Err ";
        print_int x;print_string " ";
        print_int y; print_newline() ; 0

  let col = 5
  let lig = 5

(* draw_background : unit -> unit  *)
(* draw the screen background  *)
  let draw_background () =
    Graphics.clear_graph() ;
    Graphics.set_color cBlue ;
    Graphics.fill_rect bounds bounds width (height-top_offset)

(* draw_places : unit -> unit  *)
(* draw the pieces at the start of the game *)
  let draw_places () =
    for l = 0 to 17 do
      let cp = corresp l in
      if cp <> (0,0) then
        begin
          Graphics.set_color cBlack ;
          Graphics.draw_circle
            ((fst cp)*30 + virtual_table_xoffset)
            (height - ((snd cp)*55 + 25)-50) (brightness+1) ;
          Graphics.set_color cGray ;
          Graphics.fill_circle
            ((fst cp)*30 + virtual_table_xoffset)
            (height - ((snd cp)*55 + 25)-50) brightness
        end

    done

(* draw_force_lines : unit -> unit  *)
(* draws ley-lines  *)
  let draw_force_lines () =
    Graphics.set_color cYellow ;
    let lst = [((2,1),(6,1)); ((1,2),(7,2)); ((0,3),(8,3));
               ((-1,4),(9,4)); ((0,5),(8,5)); ((5,0),(1,4));
               ((7,0),(2,5)); ((8,1),(4,5)); ((9,2),(6,5));
               ((10,3),(8, 5)); ((3,6),(1,4)); ((5,6),(2,3));
```

```
              ((7,6),(3,2)); ((9,6),(4,1)); ((10,5),(6,1))] in
    let rec lines l =
      match l with
        [] → ()
      | h :: t → let deb = fst h and complete = snd h in
        Graphics.moveto
          ((fst deb) * 30 + virtual_table_xoffset)
          (height -     ((snd deb) * 55 + 25) -50) ;
        Graphics.lineto
          ((fst complete) * 30 + virtual_table_xoffset)
          (height -     ((snd complete) * 55 + 25) -50) ;
        lines t
    in lines lst

(* draw_final_places : unit -> unit  *)
(* draws final cases for each ley-line *)
(* coordinates represent in the virtual array
used for positioning *)

let draw_final_places () =
  let lst = [(2,1); (1,2); (0,3); (-1,4); (0,5); (3,6); (5,6);
             (7,6); (9,6); (10,5); (10,3); (9,2); (8,1); (7,0);
             (5,0)] in
  let rec final l =
    match l with
      [] → ()
    | h :: t →  Graphics.set_color cBlack ;
        Graphics.draw_circle
          ((fst h)*30 + virtual_table_xoffset)
          (height - ((snd h)*55 + 25)-50) (brightness+1) ;
        Graphics.set_color cGreen ;
        Graphics.fill_circle
          ((fst h)*30 + virtual_table_xoffset)
          (height - ((snd h)*55 + 25)-50) brightness ;
        final t
  in final lst


(* draw_table : unit -> unit  *)
(* draws the whole game  *)
let draw_table () =
  Graphics.set_color cYellow ;
  draw_background () ;
  Graphics.set_line_width 5 ;
  draw_force_lines () ;
  Graphics.set_line_width 2 ;
  draw_places () ;
  draw_final_places ();
  Graphics.set_line_width 1

(* move -> couleur -> unit  *)
```

```
let draw_piece  player  (n_case,P cp)  = (*   (n_caOccup(c,v),cp) col =*)
  Graphics.set_color (if player then cBlack else cRed); (*col;*)
  let co = corresp n_case in
  let x = ((fst co)*30 + 145) and y = (height - ((snd co)*55 + 25)-50) in
  Graphics.fill_circle x y brightness ;
  Graphics.set_color cWhite ;
  Graphics.moveto (x - 3) (y - 3) ;
  let dummy = 5 in
  Graphics.draw_string (string_of_int cp) (*;*)
(*    print_string "---";print_int n_case; print_string " "; print_int cp ;print_newline() *)

(* conv : Graphics.status -> int  *)
(* convert a mouse click into a position on a virtual table permitting *)
(* its drawing *)
let conv st =
  let xx = st.Graphics.mouse_x and yy = st.Graphics.mouse_y in
  let y = (yy+10)/virtual_case_size - 6 in
  let dec =
    if y = ((y/2)*2) then 60 else 40 in
  let offset = match (-1*y) with
    0 → -2
  | 1 → -1
  | 2 → -1
  | 3 → 0
  | 4 → -1
  | _ → 12 in
  let x = (xx+dec)/virtual_case_size - 3 + offset in
  (-1*y, x)

(* line_number_to_aff : int -> int*int *)
(* convert a line number into a polition on the virtual table serving *)
(* for drawing *)
(* the coordinate returned corresponds to the final case for the line *)
let line_number_to_aff n =
  match n with
    0 → (2,1)
  | 1 → (1,2)
  | 2 → (0,3)
  | 3 → (-1,4)
  | 4 → (0,5)
  | 5 → (5,0)
  | 6 → (7,0)
  | 7 → (8,1)
  | 8 → (9,2)
  | 9 → (10,3)
  | 10 → (3,6)
  | 11 → (5,6)
  | 12 → (7,6)
  | 13 → (9,6)
  | 14 → (10,5)
  | _ → failwith "line" (*raise Rep.Out_of_bounds*)
```

```
  (* draw_lines_won : game -> unit  *)
  (* position a marker indicating the player which has taken the line *)
  (* this is done for all lines *)
  let  drawb  l  i =
    match l with
      None → failwith "draw"
    | M j  → let pos = line_number_to_aff i in
  (*               print_string "'''''";
  print_int i;
  print_string "---";
  Printf.printf "%d,%d\n" (fst pos) (snd pos);
*)                 Graphics.set_color (if j then cBlack else cRed);
      Graphics.fill_rect ((fst pos)*30 + virtual_table_xoffset-bounds)
        (height - ((snd pos)*55 + 25)-60) 20 40

  let draw_lines_won om nm  =
    for i=0 to 14 do
      if om.(i) <> nm.(i) then drawb nm.(i) i
    done
  (*********************
  let black_lines = Rep.lines_won_by_player mat Rep.Noir and
  red_lines = Rep.lines_won_by_player mat Rep.Rouge
  in
  print_string "black : "; print_int (Rep.list_size black_lines);
  print_newline () ;
  print_string "red : "; print_int (Rep.list_size red_lines);
  print_newline() ;

  let rec draw l col =
  match l with
  [] -> ()
  | h::t -> let pos = line_number_to_aff h in
  Graphics.set_color col ;
  Graphics.fill_rect ((fst pos)*30 + virtual_table_xoffset-bounds)
  (height - ((snd pos)*55 + 25)-60) 20 40 ;
  draw t col
  in draw black_lines cBlack ;
  draw red_lines cRed
  ************************************************)

  (* draw_poss : item list -> int -> unit  *)
  (* draw the pieces available for a player based on a list *)
  (* the parameter "off" indicates the position at which to place the list *)
  let draw_poss player lst off =
    let c = ref (1) in
    let rec draw l =
      match l with
        [] → ()
      | v :: t → if player  then Graphics.set_color cBlack
      else Graphics.set_color cRed;
          let x = off   and
```

```
            y = 0+(!c)*50   in
        Graphics.fill_circle x y brightness ;
        Graphics.set_color cWhite ;
        Graphics.moveto (x - 3) (y - 3) ;
        Graphics.draw_string (string_of_int v) ;
        c := !c + 1 ;
        draw t

  in draw (List.map (function P x → x) lst)

(* draw_choice : game -> unit *)
(* draw the list of pieces still available for each player *)
let draw_choice (J (ca,ma,r1,r2)) =
  Graphics.set_color cBlue ;
  Graphics.fill_rect (choice_black_offset-30) 10 60
    (height - (top_offset + bounds)) ;
  Graphics.fill_rect (choice_red_offset-30) 10 60
    (height - (top_offset + bounds)) ;
  draw_poss true r1 choice_black_offset ;
  draw_poss false r2 choice_red_offset

(* wait_click : unit -> unit *)
(* wait for a mouse click *)
let wait_click () = Graphics.wait_next_event [Graphics.Button_down]


(* item list -> item  *)
(* return, for play, the piece chosen by the user *)
let select_pion player  lst =
  let ok = ref false and
      choice = ref 99 and
      pion = ref (P(-1))
  in
  while not !ok do
    let st = wait_click () in
    let size = List.length lst in
    let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y in
    choice := (y+25)/50 - 1 ;
    if !choice <= size && ( (player && x < 65 )
                        || ( (not player) && (x > 535))) then ok := true
    else ok := false ;
    if !ok then
      try
        pion := (List.nth  lst !choice) ;
        Graphics.set_color cGreen ;
        Graphics.set_line_width 2 ;
        Graphics.draw_circle
          (if player then choice_black_offset else choice_red_offset)
          ((!choice+1)*50) (brightness + 1)
      with _  → ok := false ;
  done ;
```

```
    !pion

  (* choiceH : game -> move  *)
  (* return a move for the human player.
  return the choice of the number, the case, and the piece *)
  let rec choice player game = match game with (J(ca,ma,r1,r2)) →
    let choice = ref (P(-1))
    and c = ref (-1, P(-1)) in
    let lcl = legal_moves player game in
    while not (List.mem !c lcl) do
      print_newline();print_string "CHOICE";
      List.iter (fun (c,P p) → print_string "["; print_int c;print_string " ";
        print_int p;print_string "]")
        (legal_moves player game);
      draw_choice game;
      choice :=  select_pion player  (if player then r1 else r2) ;
  (*       print_string "choice "; print_piece !choice;*)
      c :=  (corresp2  (conv (wait_click())), !choice)
  (*       let (x,y) = !c in
   (print_string "...";print_int x; print_string " "; print_piece y;
   print_string " -> ";
   print_string "END_CHOICE";print_newline())
*)     done ;
    !c (* case, piece *)


  (* home : unit -> unit  *)
  (* place a message about the game *)
  let home () =
    Graphics.open_graph
      (" " ^ (string_of_int (width + 10)) ^ "x" ^ (string_of_int (height + 10))
       ^ "+50+50") ;
    Graphics.moveto (height / 2) (width / 2) ;
    Graphics.set_color cBlue ;
    Graphics.draw_string "Stonehenge" ;
    Graphics.set_color cBlack ;
    Graphics.moveto 2 2 ;
    Graphics.draw_string "Mixte Projets Maîtrise & DESS GLA" ;
    wait_click () ;
    Graphics.clear_graph ()

  (* exit : unit -> unit *)
  (* close everything ! *)
  let exit () =
    Graphics.close_graph ()

  (* draw_button : int -> int -> int -> int -> string -> unit  *)
  (* draw a button with a message *)
  let draw_button x y w h s =
    Graphics.set_line_width 1 ;
    Graphics.set_color cBlack ;
    Graphics.moveto x y ;
```

```
  Graphics.lineto x (y+h) ;
  Graphics.lineto (x+w) (y+h) ;
  Graphics.lineto (x+w) y ;
  Graphics.lineto x y ;
  Graphics.moveto (x+bounds) (height - (top_offset/2)) ;
  Graphics.draw_string s

(* draw_message : string -> unit  *)
(* position a message *)
let draw_message s =
  Graphics.set_color cBlack;
  Graphics.moveto 3 (height - (top_offset/2)) ;
  Graphics.draw_string s

(* erase_message : unit -> unit  *)
(* as the name indicates *)
let erase_message () =
  Graphics.set_color Graphics.white;
  Graphics.fill_rect 0 (height-top_offset+bounds) width top_offset

(* question : string -> bool  *)
(* pose the user a question, and wait for a yes/no response *)
let question s =
  let xb1 = (width/2) and xb2 = (width/2 + 30) and wb = 25 and hb = 16
  and yb = height - 20  in
  let rec attente () =
    let e = wait_click () in
    if (e.Graphics.mouse_y < (yb+hb)) & (e.Graphics.mouse_y > yb) then
      if (e.Graphics.mouse_x > xb1) & (e.Graphics.mouse_x < (xb1+wb)) then
        true
      else
        if (e.Graphics.mouse_x > xb2) & (e.Graphics.mouse_x < (xb2+wb)) then
          false
        else
          attente()
    else
      attente () in
  draw_message s;
  draw_button xb1 yb wb hb "yes";
  draw_button xb2 yb wb hb "no";
  attente()

(* q_begin : unit -> bool *)
(* Ask if the player wishes to be the first player or not *)
let q_begin () =
  let b = question "Would you like to play first ?" in
  erase_message();
  b

(* q_continue : unit -> bool *)
(* Ask if the user wishes to play the game again *)
```

```
  let q_continue () =
    let b = question "Play again ?" in
    erase_message();
    b
  (* won : unit -> unit  *)
  (* a message indicating the machine has won *)
  let won () = draw_message "I won :-)"; wait_click(); erase_message()

  (* lost : unit -> unit  *)
  (* a message indicating the machine has lost *)
  let lost () = draw_message "You won :-("; wait_click(); erase_message()

  (* nil : unit -> unit  *)
  (* a message indicating stalemate *)
  let nil () = draw_message "Stalemate"; wait_click(); erase_message()

  (* init : unit -> unit *)
  (* draw the initial game board *)
  let init () = let game = game_start () in
  draw_table () ;
  draw_choice game

  (* drawH : move -> game -> unit *)
  (* draw a piece for the human player *)
  (* let drawH cp j =  draw_piece cp cBlack ;
   draw_lines_won j
*)
  (* drawM : move -> game -> unit *)
  (* draw a piece for the machine player *)
  (* let drawM cp j =  draw_piece cp cRed ;
   draw_lines_won j
*)
  let print_placement m = match m with
    None → print_string "None "
  | M j  → print_string ("Pl "^(if j then "1 " else "2 "))

  let position player move
      (J(ca1,m1,r11,r12))
      (J(ca2,m2,r21,r22) as new_game) =
    draw_piece player move;
    draw_choice new_game;
  (*  print_string "_______OLD___________________\n";
   Array.iter print_placement m1; print_newline();
   List.iter print_piece r11; print_newline();
   List.iter print_piece r12; print_newline();
   print_string "_______NEW___________________\n";
   Array.iter print_placement m2; print_newline();
   List.iter print_piece r21; print_newline();
   List.iter print_piece r22; print_newline();
*)  draw_lines_won m1 m2

  (*
```

```
   if player then draw_piece move cBlack
   else  draw_piece move cRed
*)
  let q_player () =
    let b = question "Is there a machine playing?" in
    erase_message ();
    b
  end;;
Characters 11114-11127:
Warning: this expression should have type unit.
Characters 13197-13209:
Warning: this expression should have type unit.
Characters 13345-13357:
Warning: this expression should have type unit.
Characters 13478-13490:
Warning: this expression should have type unit.
module Stone_graph :
  sig
    type piece = Stone_rep.piece
    and placement = Stone_rep.placement
    and case = Stone_rep.case
    and game = Stone_rep.game
    and move = Stone_rep.move
    val brightness : int
    val cBlack : Graphics.color
    val cRed : Graphics.color
    val cYellow : Graphics.color
    val cGreen : Graphics.color
    val cWhite : Graphics.color
    val cGray : Graphics.color
    val cBlue : Graphics.color
    val width : int
    val height : int
    val top_offset : int
    val bounds : int
    val virtual_table_xoffset : int
    val choice_black_offset : int
    val choice_red_offset : int
    val virtual_case_size : int
    val corresp : int -> int * int
    val corresp2 : int * int -> int
    val col : int
    val lig : int
    val draw_background : unit -> unit
    val draw_places : unit -> unit
    val draw_force_lines : unit -> unit
    val draw_final_places : unit -> unit
    val draw_table : unit -> unit
    val draw_piece : bool -> int * Stone_rep.piece -> unit
    val conv : Graphics.status -> int * int
    val line_number_to_aff : int -> int * int
    val drawb : Stone_rep.placement -> int -> unit
```

```
    val draw_lines_won :
      Stone_rep.placement array -> Stone_rep.placement array -> unit
    val draw_poss : bool -> Stone_rep.piece list -> int -> unit
    val draw_choice : Stone_rep.game -> unit
    val wait_click : unit -> Graphics.status
    val select_pion : bool -> Stone_rep.piece list -> Stone_rep.piece
    val choice : bool -> Stone_rep.game -> int * Stone_rep.piece
    val home : unit -> unit
    val exit : unit -> unit
    val draw_button : int -> int -> int -> int -> string -> unit
    val draw_message : string -> unit
    val erase_message : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val print_placement : Stone_rep.placement -> unit
    val position :
      bool ->
      int * Stone_rep.piece -> Stone_rep.game -> Stone_rep.game -> unit
    val q_player : unit -> bool
  end
```

Assembly. We thus write module `Stone_graph` which describes a graphical interface compatible with signature `DISPLAY`. We construct `Stone_skeletonG` similar to `C4_skeletonG`, passing in the arguments appropriate for the Stonehenge game, applying the parametric module `FSkeleton`.

```
# module Stone_skeletonG = FSkeleton (Stone_rep)
                                     (Stone_graph)
                                     (Stone_eval)
                                     (FAlphabeta (Stone_rep) (Stone_eval)) ;;
module Stone_skeletonG :
  sig
    val depth : int ref
    exception Won
    exception Lost
    exception Nil
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val again : unit -> bool
    val play_game : Stone_graph.game ref
    val exit : unit -> unit
    val home : unit -> unit
    val playH : bool -> unit -> unit
```

```
    val playM : bool -> unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
  end
```

We may thus construct the principal module `Stone_mainG`.

```
# module Stone_mainG = FMain(Stone_skeletonG) ;;
module Stone_mainG :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

Launching `Stone_mainG.main ()` opens the window shown in figure 17.6. After displaying a dialogue to show who is playing, the game begins. A human player will select a piece and place it.

To Learn More

This organization of these applications involves using several parametric modules that permit direct reuse of `FAlphabeta` and `FSkeleton` for the two games we have written. With Stonehenge, some of the functions from `Stone_rep`, needed for `play`, which do not appear in `REPRESENTATION`, are used by the evaluation function. That is why the module `Stone_rep` was not closed immediately by `REPRESENTATION`. This partitioning of modules for the specific aspects of games allows incremental development without making the game schema dependencies (presented in figure 17.4) fragile.

A first enhancement involves games where given a position and a move, it is easy to determine the preceding position. In such cases, it may be more efficient to not bother making a copy of the game for function `play`, but rather to conserve a history of moves played to allow *backtracking*. This is the case for Connect 4, but not for Stonehenge.

A second improvement is to capitalize on a player's response time by evaluating future positions while the other player is selecting his next move. For this, one may use threads (see chapter 19), which allow concurrent calculation. If the player's response is one that has already been explored, the gain in time will be immediate, if not we must start again from the new position.

A third enhancement is to build and exploit dictionaries of opening moves. We have been able to do so with Stonehenge, but it is also useful for many other games where the set of legal moves to explore is particularly large and complex at the start of the game. There is much to be gained from estimating and precalculating some "best" moves from the starting positions and retaining them in some sort of database. One may add a bit of "spice" (and perhaps unpredictability) to the games by introducing an element of chance, by picking randomly from a set of moves with similar or identical values.

A fourth view is to not limit the search depth to a fixed depth value, but rather to limit the search by a calculation time period that is not to be exceeded. In this manner, the program will be able to efficiently search to deeper depths when the number of remaining moves becomes limited. This modification requires slight modification to `minmax` in order to be able to re-examine a tree to increase its depth.

A game-dependent heuristic, parameterized by `minmax`, may be to choose which branches in the search should be pursued and which may be quickly abandoned.

There are also many other games that require little more than to be implemented or reimplemented. We might cite many classic games: Checkers, Othello, Abalone, ..., but also many lesser-known games that are, nevertheless, readily playable by computer. You may find on the web various student projects including Checkers or the game Nuba.

Link: http://www.gamecabinet.com/rules/Nuba.html

Games with stochastic qualities, such as card games and dice games, necessitate a modification of the minimax-$\alpha\beta$ algorithm in order to take account of the probabilities of the selections.

We will return to the interfaces of games in chapter 21 in constructing web-based interfaces, providing without further cost the ability to return to the last move. This also allows further benefits from the modular organization that allows modifying no more than just an element, here the game state and interactions, to extend the functionality to support two player games.

Fancy Robots

The example in this section illustrates the use of objects from the graphics library. We will revisit the concepts of simple inheritance, overriding methods and dynamic dispatch. We also see how parametric classes may be profitably used.

The application recognizes two principal categories of objects: a world and robots. The world represents a state space within which the robots evolve. We will have various classes of robots, each possessing its own strategy to move around in the world.

The principle of interaction between robots and the world here is extremely simple. The world is completely in control of the game: it asks, turn by turn, each of the robots if they know their next position. Each robot determines its next position fairly blindly. They do not know the geometry of the world, nor what other robots may be present. If the position requested by a robot is legal and open, the world will place the robot at that position.

The world displays the evolution of the robots via an interface. The (relative) complexity of the conception and development of this example is in the always-necessary separation between a behavior (here the evolution of the robots) and its interface (here the tracking of this evolution).

General Description The application is developed in two stages.

1. A group of definitions providing pure calculation classes for the world and for the diverse set of envisaged robots.
2. A group of definitions using the preceding set, adding whatever is necessary to add in an interface.
   We provide two examples of such interfaces: a rudimentary text-based interface, and a more elaborate one using a graphical library.

In the first section, we provide the *abstract* definitions for the robots. Then (page 553), we provide the pure abstract definition for the world. In the next section (page 554), we introduce the text interface for the robots, and in the fourth section (page 556), the interface for the world. On page 559 we introduce a graphical interface for the robots and finally (page 562) we define a world for the graphical interface.

"Abstract" Robots

The first thing to do is to examine robots abstractly, independent of any consideration of the environment in which they will move, that is to say, the interface that displays them.

```
# class virtual robot (i0:int) (j0:int) =
    object
      val mutable i = i0
      val mutable j = j0
      method get_pos = (i,j)
      method set_pos (i', j') = i <- i'; j <- j'
      method virtual next_pos : unit → (int * int)
    end ;;
```

A robot is an entity which knows, or believes it knows, its position (`i` and `j`), is capable of communicating that position to a requester (`get_pos`), is able to modify this knowledge if it knows precisely where it should be (`set_pos`) and may decide to move towards a new position (`next_pos`).

To improve the readability of the program, we define relative movements based on absolute directions:

```
# type dir = North | East | South | West | Nothing  ;;

# let walk (x,y) = function
      North → (x,y+1) | South → (x,y-1)
    | West  → (x-1,y) | East  → (x+1,y)
    | Nothing → (x,y) ;;
val walk : int * int -> dir -> int * int = <fun>

# let turn_right = function
```

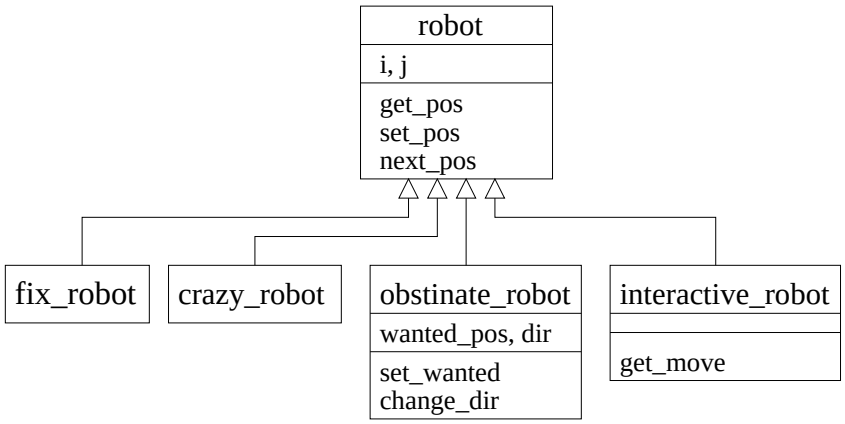


Figure 17.9: Hierarchy of pure robot classes

```
    North → East | East → South | South → West | West → North | x → x ;;
val turn_right : dir -> dir = <fun>
```

The schema is shown by the virtual class `robots` from which we define four distinct species of robots (see figure 17.9) to more precisely see their manner of motion:

- Fixed robots which never move:

```
# class fix_robot i0 j0 =
    object
      inherit robot i0 j0
      method next_pos() = (i,j)
    end ;;
```

- Crazy robots which move at random:

```
# class crazy_robot i0 j0 =
    object
      inherit robot i0 j0
      method next_pos () = ( i+(Random.int 3)-1 , j+(Random.int 3)-1 )
    end ;;
```

- Obstinate robots which keep trying to advance in one direction whenever they are able to do so,

```
# class obstinate_robot i0 j0 =
    object(self)
      inherit robot i0 j0
      val mutable wanted_pos = (i0,j0)
      val mutable dir = West

      method private set_wanted_pos d = wanted_pos <- walk (i,j) d
      method private change_dir = dir <- turn_right dir
```

```
      method next_pos () = if (i,j) = wanted_pos
        then let np = walk (i,j) dir in ( wanted_pos <- np ; np )
        else ( self#change_dir ; wanted_pos <- (i,j) ; (i,j) )
    end ;;
```

- Interactive robots which obey the commands of an exterior operator:

```
# class virtual interactive_robot i0 j0 =
    object(self)
      inherit robot i0 j0
      method virtual private get_move : unit → dir
      method next_pos () = walk (i,j) (self#get_move ())
     end ;;
```

The case of the interactive robot is different from the others in that its behavior is controlled by an interface that permits communicating orders to it. To deal with this, we provide a virtual method to communicate this order. As a consequence, the class `interactive_robot` remains abstract.

Note that not only do the four specialized robot classes inherit from class `robot`, but also any others that have the same type. In effect, the only methods that we have added are the private methods that therefore do not appear in the type signatures of the instances of these classes (see page 449). This property is indispensable if we wish to consider all the robots to be objects of the same type.

Pure World

A *pure* world is a world that is independent of an interface. It is understood as the state space of positions which a robot may occupy. It takes the form of a grid of size `l` × `h`, with a method `is_legal` to assure that a coordinate is a valid position in the world, and a method `is_free` indicates whether or not a robot occupies a given position.

In practice, a world manages the list of `robots` present on its surface while a method, `add`, allows new robots to enter the world.

Finally, a world is made visible by the method `run`, allowing the world to come to life.

```
# class virtual ['robot_type] world (l0:int) (h0:int) =
    object(self)
      val l = l0
      val h = h0
      val mutable robots =  ( [] : 'robot_type list )
      method add r = robots <- r :: robots
      method is_free p = List.for_all (fun r → r#get_pos <> p)  robots
      method virtual is_legal : (int * int) → bool

      method private run_robot r =
        let p = r#next_pos ()
```

```
        in if (self#is_legal p) & (self#is_free p) then r#set_pos p

      method run () =
        while true do List.iter (function r → self#run_robot r) robots done
    end ;;
class virtual ['a] world :
  int ->
  int ->
  object
    constraint 'a =
      < get_pos : int * int; next_pos : unit -> int * int;
        set_pos : int * int -> unit; .. >
    val h : int
    val l : int
    val mutable robots : 'a list
    method add : 'a -> unit
    method is_free : int * int -> bool
    method virtual is_legal : int * int -> bool
    method run : unit -> unit
    method private run_robot : 'a -> unit
  end
```

The Objective Caml type system does not permit leaving the types of robots undetermined (see page 460). To resolve this problem, we might consider restraining the type to those of the class `robot`. But that would forbid populating a world with objects other than those having exactly the same type as `robot`. As a result, we have instead decided to parameterize `world` with the type of the robots that populate it. We may thereby instantiate this type parameter with textual robots or graphical robots.

Textual Robots

Text Objects To obtain robots controllable via a textual interface, we define a class of text objects (`txt_object`).

```
# class txt_object (s0:string) =
    object
      val name = s0
       method get_name = name
    end ;;
```

An Interface Class: Abstract Textual Robots By double inheritance from `robots` and `txt_object`, we obtain the abstract class `txt_robot` of textual robots.

```
# class virtual txt_robot i0 j0 =
    object
```

```
      inherit robot i0 j0
      inherit txt_object "Anonymous"
   end ;;
class virtual txt_robot :
  int ->
  int ->
  object
    val mutable i : int
    val mutable j : int
    val name : string
    method get_name : string
    method get_pos : int * int
    method virtual next_pos : unit -> int * int
    method set_pos : int * int -> unit
  end
```

This class defines a world with a textual interface (see page 556). The inhabitants of this world will not be objects of `txt_robot` (since this class is abstract) nor inheritors of this class. The class `txt_robot` is, in a way, an *interface classe* permitting the compiler to identify the method types (calculations and interfaces) of the inhabitants of the text interface world. The use of such a specification class provides the separation we wish to maintain between calculations and interface.

Concrete Text Robots These are simply obtained via double inheritance; figure 17.10 shows the hierarchy of classes.

```
# class fix_txt_robot i0 j0 =
    object
      inherit fix_robot i0 j0
      inherit txt_object "Fix robot"
    end ;;

# class crazy_txt_robot i0 j0 =
    object
      inherit crazy_robot i0 j0
      inherit txt_object "Crazy robot"
    end ;;

# class obstinate_txt_robot i0 j0 =
    object
      inherit obstinate_robot i0 j0
      inherit txt_object "Obstinate robot"
    end ;;
```

The interactive robots require, for a workable implementation, defining their method of interacting with the user.

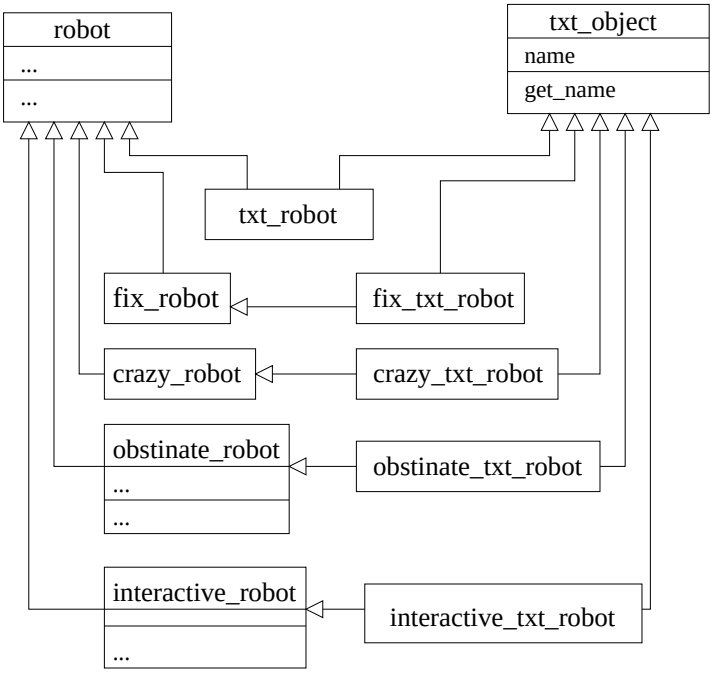


Figure 17.10: Hierarchy of classes for text mode robots

```
# class interactive_txt_robot i0 j0 =
    object
      inherit interactive_robot i0 j0
      inherit txt_object "Interactive robot"
      method private get_move () =
        print_string "Which dir : (n)orth (e)ast (s)outh (w)est ? ";
        match read_line() with
            "n" → North  | "s" → South
          | "e" → East   | "w" → West
          |  _  → Nothing
    end ;;
```

Textual World

The text interface world is derived from the pure world by:

1. Inheritance from the generic class `world` by instantiating its type parameter with the class specified by `txt_robot`, and
2. Redefinition of the method `run` to include the different textual methods.

```
# class virtual txt_world (l0:int) (h0:int) =
    object(self)
      inherit [txt_robot] world l0 h0 as super

      method private display_robot_pos r =
        let (i,j) = r#get_pos in Printf.printf "(%d,%d)" i j

      method private run_robot r =
        let p = r#next_pos ()
        in if (self#is_legal p) & (self#is_free p)
          then
            begin
              Printf.printf "%s is moving from " r#get_name ;
              self#display_robot_pos r ;
              print_string " to " ;
              r#set_pos p;
              self#display_robot_pos r ;
            end
          else
            begin
              Printf.printf "%s is staying at " r#get_name ;
              self#display_robot_pos r
            end ;
          print_newline () ;
          print_string"next - ";
          ignore (read_line())

      method run () =
        let print_robot r =
          Printf.printf "%s is at " r#get_name ;
          self#display_robot_pos r ;
          print_newline ()
        in
          print_string "Initial state :\n";
          List.iter print_robot robots;
          print_string "Running :\n";
          super#run()  (* 1 *)
    end ;;
```

We direct the reader's attention to the call to `run` of the ancestor class (this method call is marked `(* 1 *)` in the code) in the redefinition of the same method. There we have an illustration of the two possible types of method dispatch: static or dynamic (see page 446). The call to *super#run* is static. This is why we name the superclass: to be able to call the methods when they are redefined. On the other hand, in *super#run* we find a call to *self#run_robot*. This is a dynamic dispatch; the method defined in class `txt_world` is executed, not that of `world`. Were the method from `world` executed, nothing would be displayed, and the method in `txt_world` would remain useless.

The planar rectangular text world is obtained by implementing the final method that still remains abstract: `is_legal`.

```
# class closed_txt_world l0 h0 =
    object(self)
      inherit txt_world l0 h0
      method is_legal (i,j) = (0<=i) & (i<l) & (0<=j) & (j<h)
    end ;;
```

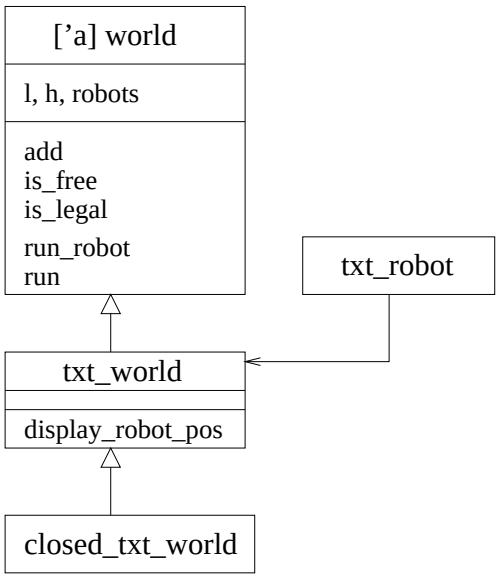


Figure 17.11: Hierarchy of classes in the textual planar rectangular world

We may proceed with a small essay in typing:

```
let w = new closed_txt_world 5 5
and r1 = new fix_txt_robot 3 3
and r2 = new crazy_txt_robot 2 2
and r3 = new obstinate_txt_robot 1 1
and r4 = new interactive_txt_robot 0 0
in w#add r1; w#add r2; w#add r3; w#add r4; w#run () ;;
```

We may skip, for the moment, the implementation of a graphical interface for our world of robots. In due course, we will obtain an application having an appearance like figure 17.12.

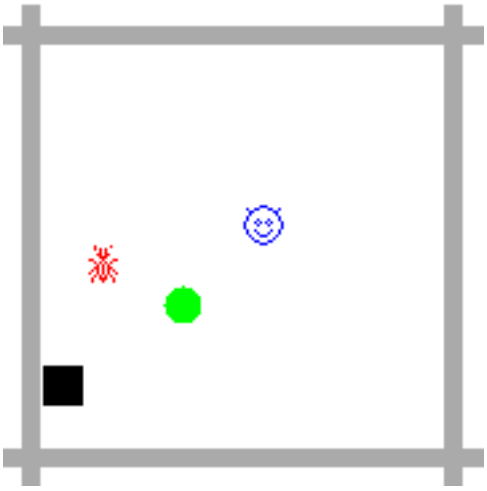


Figure 17.12: The graphical world of robots

Graphical Robots

We may implement robots in a graphical mode by following the same approach as with the text mode:

1. define a generic graphical object,
2. define an abstract class of graphical robots by double inheritance from robots and graphical objects (analogous to the interface class of page 554),
3. define, through double inheritance, the particular behavior of robots.

Generic Graphical Objects

A simple graphical object is an object possessing a `display` method which takes, as arguments, the coordinates of a pixel and displays it.

```
# class virtual graph_object =
    object
      method virtual display : int → int → unit
    end ;;
```

From this specification, it would be possible to implement graphical objects with extremely complex behavior. We will content ourselves for now with a class `graph_item`, displaying a bitmap that serves to represent the object.

```
# class graph_item x y im =
    object (self)
```

```
      val size_box_x = x
      val size_box_y = y
      val bitmap = im
      val mutable last = None

      method private erase = match last with
      Some (x,y,img) → Graphics.draw_image img x y
        | None → ()

      method private draw i j = Graphics.draw_image bitmap i j
      method private keep i j =
        last <- Some (i,j,Graphics.get_image i j size_box_x size_box_y) ;

      method display i j = match last with
      Some (x,y,img) → if x<>i || y<>j
        then ( self#erase ; self#keep i j ; self#draw i j )
        | None → ( self#keep i j ; self#draw i j )
    end ;;
```

An object of graph_item stores the portion of the image upon which it is drawn in order to restore it in subsequent redraws. In addition, if the image has not been moved, it will not be redrawn.

```
# let foo_bitmap =  [|[| Graphics.black |]|] ;;
# class square_item x col =
    object
      inherit graph_item x x (Graphics.make_image foo_bitmap)
      method private draw i j =
        Graphics.set_color col ;
        Graphics.fill_rect (i+1) (j+1) (x-2) (x-2)
    end ;;

# class disk_item r col =
    object
      inherit graph_item (2*r) (2*r) (Graphics.make_image foo_bitmap)
      method private draw i j =
        Graphics.set_color col ;
        Graphics.fill_circle (i+r) (j+r) (r-2)
    end ;;

# class file_bitmap_item name =
    let ch = open_in name
    in let x = Marshal.from_channel ch
    in let y = Marshal.from_channel ch
    in let im = Marshal.from_channel ch
    in let () = close_in ch
    in object
    inherit graph_item x y (Graphics.make_image im)
       end ;;
```

We specialize the `graph_item` with instances of crosses, disks, and other bitmaps, read from a file.

The abstract graphical robot is both a robot and a graphical object.

```
# class virtual graph_robot i0 j0 =
    object
      inherit robot i0 j0
      inherit graph_object
    end ;;
```

Graphical robots that are fixed, crazy, and obstinate are specialized graphical objects.

```
# class fix_graph_robot i0 j0 =
    object
      inherit fix_robot i0 j0
      inherit disk_item 7 Graphics.green
    end ;;

# class crazy_graph_robot i0 j0 =
    object
      inherit crazy_robot i0 j0
      inherit file_bitmap_item "crazy_bitmap"
     end ;;

# class obstinate_graph_robot i0 j0 =
    object
      inherit obstinate_robot i0 j0
      inherit square_item 15 Graphics.black
    end ;;
```

The interactive graphical robot uses the primitives `key_pressed` and `read_key` of module `Graphics` to determine its next move. We again see the key presses `8`, `6`, `2` and `4` on the numeric keypad (`NumLock` button active). In this manner, the user is not obliged to provide direction at each step in the simulation.

```
# class interactive_graph_robot i0 j0 =
    object
      inherit interactive_robot i0 j0
      inherit file_bitmap_item "interactive_bitmap"
      method private get_move () =
        if not (Graphics.key_pressed ()) then Nothing
        else match Graphics.read_key() with
```

```
        '8' → North | '2' → South | '4' → West  | '6' → East | _ → Nothing
    end ;;
```

Graphical World

We obtain a world with a graphical interface by inheriting from the pure world, instantiating the parameter `'a_robot` with the graphical robot abstract class `graph_robot`. As with the text mode world, the graphical world provides its own method, `run_robot`, to implement the robot's behavior as well as the general activation method `run`.

```
# let delay x = let t = Sys.time () in while (Sys.time ()) -. t < x do () done ;;

# class virtual graph_world l0 h0 =
    object(self)
      inherit [graph_robot] world l0 h0 as super
      initializer
        let gl = (l+2)*15 and  gh = (h+2)*15 and lw=7 and cw=7
        in Graphics.open_graph (" "^(string_of_int gl)^"x"^(string_of_int gh)) ;
           Graphics.set_color (Graphics.rgb 170 170 170) ;
           Graphics.fill_rect 0 lw gl lw ;
           Graphics.fill_rect (gl-2*lw) 0 lw gh ;
           Graphics.fill_rect 0 (gh-2*cw) gl cw ;
           Graphics.fill_rect lw 0 lw gh

      method run_robot r = let p = r#next_pos ()
                           in delay 0.001 ;
                              if (self#is_legal p) & (self#is_free p)
                              then ( r#set_pos p ; self#display_robot r)

      method display_robot r = let (i,j) = r#get_pos
                               in r#display (i*15+15) (j*15+15)

      method run() = List.iter self#display_robot robots ;
                     super#run()
    end ;;
```

Note that the graphical window is created at the time that an object of this class is initialized.

The rectangular planar graphical world is obtained in much the same manner as with the rectangular planar textual world.

```
# class closed_graph_world l0 h0 =
    object(self)
      inherit graph_world l0 h0
```

```
      method is_legal (i,j) = (0<=i) & (i<l) & (0<=j) & (j<h)
    end ;;
class closed_graph_world :
  int ->
  int ->
  object
    val h : int
    val l : int
    val mutable robots : graph_robot list
    method add : graph_robot -> unit
    method display_robot : graph_robot -> unit
    method is_free : int * int -> bool
    method is_legal : int * int -> bool
    method run : unit -> unit
    method run_robot : graph_robot -> unit
  end
```

We may then test the graphical application by typing in:

```
let w = new closed_graph_world 10 10 ;;
w#add (new fix_graph_robot 3 3) ;;
w#add (new crazy_graph_robot 2 2) ;;
w#add (new obstinate_graph_robot 1 1) ;;
w#add (new interactive_graph_robot 5 5) ;;
w#run () ;;
```

To Learn More

The implementation of the method `run_robot` in different worlds suggests that the robots are potentially able to move to any point on the world the moment it is empty and legal. Unfortunately, nothing prevents a robot from modifying its position arbitrarily; the world cannot prevent it. One remedy would consist of having robot positions being controlled by the world; when a robot attempts to move, the world verifies not only that the new position is legal, but also that it constitutes an authorized move. In that case, the robot must be capable of asking the world its actual position, with the result that the robot class must become dependent on the world's class. The robot class would take, as a type parameter, the world class.

This modification permits defining robots capable of querying the world in which they run, thus behaving as dependents of the world. We may then implement robots which follow or avoid other robots, try to block them, and so forth.

Another extension would be to permit robots to communicate with one another, exchanging information, perhaps constituting themselves into teams of robots.

The chapters of the next section allow making execution of robots independent from one another: by making use of `Threads` (see page 599), each may execute as a distinct

process. They may profit from the possibilities of distributed computing (see 623) where the robots become *clients* executing on remote machines that announce their movements or request other information from a world that behaves as a *server*. This problem is dealt with on page 656.