3.11 *Extended example:* Solving games

Using computers to solve puzzle-like games illustrates the power of computing machinery, or at least an illustration of what machinery can do better than the human mind can. Games that have positions or configurations—that is, the arrangement of pieces on a board or numbers or letters in a puzzle—can be played by a computer by sifting through large amounts of data on possible outcomes of the game. The speed and abundance of computer memory gives machines an incomparable advantage over humans. An unbeatable tic-tac-toe-playing program is fairly straightforward to write. Computer chess-playing engines now regularly beat the world's top human players.

In this section, we try our hand at writing programs to solve puzzles that require searching through solutions. The first is called *bulls and cows*. One player thinks of a four-digit number with no digit appearing more than once (we also assume that the number does not begin with 0). The other player guesses the number. After each guess, the first player tells how many *bulls* and *cows* the guess has: a digit in the correct place is a bull, and a correct digit in a wrong place is a cow. If the secret number is 4721 and the guess is 5124, then this guess has one bull (2) and two cows (1 and 4).

We will represent a solution or guess in ML using a tuple of `int`s. Here is how we test if a 4-tuple is a valid guess (the function `makeNoRepeats` comes from Exercise 2.7.8):

```
fun isValidGuess(a,b,c,d) =
     0 < a andalso a < 10 andalso 0 <= b andalso b < 10 andalso
     0 <= c andalso c < 10 andalso 0 <= c andalso c < 10 andalso
     count(makeNoRepeats([a, b, c, d])) = 4;
```

First consider how to make the computer play the role of the first player and evaluate the second player's guesses. (Later we will write a program that acts as the guesser.) This program will be interactive. When the human player makes a guess, he or she will call a function that will check the guess and report on bulls and cows. To do this, we need the system to remember the secret number and the number of guesses between function calls. This is not the usual mode for ML, but it still can be done using *reference variables*. symbols that hold values that can be updated. Reference variables are declared in the ordinary way except with the keyword `ref` prepended to the value:

reference variables

```
val soln = ref (4, 7, 2, 1);
val guesses = ref 0;
```

To set a reference variable to a new value, we use the operator `:=`. Such an operation is a statement, returning a `unit` value, like `print` does. Recall from Section 1.16 that expressions and statements can be compounded using a statement list. For example, we can reset the game using the following function (realistically

we would want the computer to generate the secret number randomly, but this is just a warm-up for us):

```
fun resetGuesses() = (soln := (2, 6, 4, 9);
                      guesses := 0);
```

The function `bullsAndCows` takes a guess and the solution and returns a tuple, the number of bulls and cows. For each digit in the guess, we compare it with the digit in the same position in the solution; similarly, we see if each digit in the guess is one of the other digits in the solution. We package these as lists of booleans, which `numTrue` counts.

```
fun numTrue([]) = 0
  | numTrue(false::rest) = numTrue(rest)
  | numTrue(true::rest) = 1 + numTrue(rest);
fun bullsAndCows((a,b,c,d), (aa, bb, cc, dd)) =
    (numTrue([a = aa, b = bb, c = cc, d = dd]),
     numTrue([contains(a, [bb, cc, dd]), contains(b, [aa, cc, dd]),
              contains(c, [aa, bb, dd]), contains(d, [aa, bb, cc])]));
```

The human player submits a guess with `checkGuess`, which reports on the bulls and cows in readable format, as well as keeping track of the number of guesses.

```
fun checkGuess(a, b, c, d) =
   if isValidGuess(a, b, c, d)
   then
      let val (bulls, cows) = bullsAndCows((a, b, c, d), !soln);
      in
         (guesses := !guesses + 1;
          print("Guess # " ^ Int.toString(!guesses) ^ "\n");
          print("bulls: " ^ Int.toString(bulls) ^ "\n");
          print("cows: " ^ Int.toString(cows) ^ "\n");
          if bulls = 4 then print ("Correct!\n") else ())
      end
   else
      print("Invalid guess.\n");

- checkGuess(9,7,2,4);

Guess # 1
bulls: 2
cows: 1
val it = () : unit

- checkGuess(4,7,2,1);
```

```
Guess # 2
bulls: 4
cows: 0
Correct!
val it = () : unit
```

A more interesting problem is having the computer guess. Here is our strategy: We generate a giant list of all possible guesses. We pull guesses from this list. Every time we are told the bulls and cows for our previous guess, we remove all guesses from the list that are inconsistent with that result. Suppose we guess 2943 and are told that there is 1 bull and 2 cows. Then we would eliminate 5913 from the list because if 5913 were the answer, then our guess 2943 would have had 2 bulls and 0 cows, but we would keep 3742 because it is consistent with the bull and cow result—it a possible solution, based on this information.

Our first task is generating the initial guess list. We will create a list containing all possible combinations of four element from the set of digits. This is similar to our function to generate a powerset (Exercise 1.15.15): for each combination of two elements, we want to add each digit to it, then add each digit to all of those combinations of size three. (See Exercise 1.12.10 for `listify` and Exercise 1.15.14 for `addToEach`. The variable `guessList` is initially set to an empty list as a dummy variable; we must type it explicitly.)

```
val guessList = ref ([]:(int*int*int*int) list);
fun addEachToEach([], y) = []
  | addEachToEach(a::aRest, y) =
      addToEach(a, y)@addEachToEach(aRest, y);
fun enumerateCombinations(x, 1) = listify(x)
  | enumerateCombinations(x, r) =
     addEachToEach(x, enumerateCombinations(x, r-1));
```

But this is too much. Our guesses are supposed to be tuples, not lists, and this process generates many invalid guesses. We can remove those. Also, we would like to be able to print guesses readably.

```
fun makeGuessesTuples([]) = []
  | makeGuessesTuples([a,b,c,d]::rest) =
          (a, b, c, d)::makeGuessesTuples(rest);
fun removeInvalid([]) = []
  | removeInvalid(a::rest) =
        if isValidGuess(a) then a::removeInvalid(rest)
                           else removeInvalid(rest);
fun printGuess(a, b, c, d) =
   print(Int.toString(a) ^ " " ^ Int.toString(b) ^ " " ^
               Int.toString(c) ^ " " ^ Int.toString(d) ^ "\n");
```

Now, to generate a first guess, we enumerate our guesses and pick one. We keep track of our most recent guess in a reference variable `currentGuess`.

```
val currentGuess = ref (0,0,0,0);
fun firstGuess() =
     (guessList :=
         removeInvalid(makeGuessesTuples(
                enumerateCombinations([0,1,2,3,4,5,6,7,8,9], 4)));
      currentGuess := hd(!guessList);
      guessList := tl(!guessList);
      printGuess(!currentGuess));
```

The user will play by calling the function `guessAgain`, providing the number of bulls and cows in the previous guess. Our last task, then, is to filter our guess list by removing those guesses we now know to be infeasible. For a given guess `a`, `bullsAndCows(!currentGuess, a)` determines the number of bulls and cows there would be if `a` were the right answer and `currentGuess` were guessed; keep `a` only if it matches what the user supplied.

```
fun removeInfeasible(bulls, cows, []) = []
  | removeInfeasible(bulls, cows, a::rest) =
     if bullsAndCows(!currentGuess, a) = (bulls, cows)
     then a::removeInfeasible(bulls, cows, rest)
     else removeInfeasible(bulls, cows, rest);
fun guessAgain(bulls, cows) =
    (guessList := removeInfeasible(bulls, cows, !guessList);
     currentGuess := hd(!guessList);
     guessList := tl(!guessList);
     printGuess(!currentGuess));
```

Our secret number is 2185.

```
- firstGuess();

1 2 3 4
val it = () : unit

- guessAgain(0, 2);

2 1 5 6
val it = () : unit

- guessAgain(2, 1);
```

```
2 1 6 7
val it = () : unit

- guessAgain(2, 0);

2 1 8 5
val it = () : unit
```

All told, bulls and cows is still a simple game. Your project is a program to solve Sudoku puzzles. The rest of this section will provide the infrastructure and layout a skeleton of the algorithm.

A Sudoku puzzle contains 81 spaces laid out in a 9×9 grid, and organized not only into rows and columns but also into nine 3×3 squares. Each space must be filled with one of nine values, and each value must appear exactly once in each row, column, and square. The values for some spaces are given as part of the puzzle; to solve the puzzle, the player must fill-in the remaining spaces. Traditionally the values are the digits 1–9, but they could be any nine arbitrary values.

To model an instance of the puzzle (a Sudoku *board*), we use datatypes to define Sudoku values and Sudoku spaces. A space is either a single, given value, or it is an open space together with a list of possible values.

```
datatype sudokuValue = S1 | S2 | S3 | S4 | S5 | S6 |
                       S7 | S8 | S9;
datatype sudokuSpace = Given of sudokuValue |
                       Open of sudokuValue list;
```

Initially any open space has all values of sudokuValue in its list of possibilities. Our program to solve the puzzle will narrow down the list for each open space until each has only one possibility. So, when we print the board, for each given space and for each open space with only one possibility left, we print that one value; for each open space with more than one possibility left, we print an underscore, indicating it is still blank; if any space has no possibilities left, the board is in an impossible state, and we print an X for that space. Hence `printSpace` and `printBoard`:

```
fun printSpace(Given(a)) = printValue(a)
  | printSpace(Open([a])) = printValue(a)
  | printSpace(Open([])) = "X"
  | printSpace(Open(aa)) = "_";
fun printBoard(board) =
  let fun printHelper([], n) = ()
        | printHelper(a::rest, n) =
          (print(printSpace(a) ^ " ");
           if n = 80 then print("\n")
           else if n mod 27 = 26
```