6

Applications

The reason to prefer one programming language over another lies in the ease of developing and maintaining robust applications. Therefore, we conclude the first part of this book, which dealt with a general presentation of the Objective Caml language, by demonstrating its use in a number of applications.

The first application implements a few functions which are used to write database queries. We emphasize the use of list manipulations and the functional programming style. The user has access to a set of functions with which it is easy to write and run queries using the Objective Caml language directly. This application shows the programmer how he can easily provide the user with most of the query tools that the user should need.

The second application is an interpreter for a tiny BASIC[1]. This kind of imperative language fueled the success of the first microcomputers. Twenty years later, they seem to be very easy to design. Although BASIC is an imperative language, the implementation of the interpreter uses the functional features of Objective Caml, especially for the evaluation of commands. Nevertheless, the lexer and parser for the language use a mutable structure.

The third application is a one-player game, Minesweeper, which is fairly well-known since it is bundled with the standard installation of Windows systems. The goal of the game is to uncover a bunch of hidden mines by repeatedly uncovering a square, which then indicates the number of mines around itself. The implementation uses the imperative features of the language, since the data structure used is a two-dimensional array which is modified after each turn of the game. This application uses the `Graphics` module to draw the game board and to interact with the player. However, the automatic uncovering of some squares will be written in a more functional style.
This latter application uses functions from the `Graphics` module described in chapter

1. which means “Beginner’s All purpose Symbolic Instruction Code”.

5 (see page 117) as well as some functions from the `Random` and `Sys` modules (see chapter 8, pages 216 and 234).

Database queries

The implementation of a database, its interface, and its query language is a project far too ambitious for the scope of this book and for the Objective Caml knowledge of the reader at this point. However, restricting the problem and using the functional programming style at its best allows us to create an interesting tool for query processing. For instance, we show how to use iterators as well as partial application to formulate and execute queries. We also show the use of a data type encapsulating functional values.

For this application, we use as an example a database on the members of an association. It is presumed to be stored in the file `association.dat`.

Data format

Most database programs use a "proprietary" format to store the data they manipulate. However, it is usually possible to store the data as some text that has the following structure:

- the database is a list of *cards* separated by carriage-returns;
- each card is a list of *fields* separated by some given character, `':'` in our case;
- a field is a string which contains no carriage-return nor the character `':'`;
- the first card is the list of the names associated with the fields, separated by the character `'|'`.

The association data file starts with:

```
Num|Lastname|Firstname|Address|Tel|Email|Pref|Date|Amount
0:Chailloux:Emmanuel:Université P6:0144274427:ec@lip6.fr:email:25.12.1998:100.00
1:Manoury:Pascal:Laboratoire PPS::pm@lip6.fr:mail:03.03.1997:150.00
2:Pagano:Bruno:Cristal:0139633963::mail:25.12.1998:150.00
3:Baro:Sylvain::0144274427:baro@pps.fr:email:01.03.1999:50.00
```

The meaning of the fields is the following:

- `Num` is the member number;
- `Lastname`, `Firstname`, `Address`, `Tel`, and `Email` are obvious;
- `Pref` indicates the means by which the member wishes to be contacted: by mail (`mail`), by email (`email`), or by phone (`tel`);
- `Date` and `Amount` are the date and the amount of the last membership fee received, respectively.

We need to decide what represention the program should use internally for a database. We could use either a list of cards or an array of cards. On the one hand, a list has the nice property of being easily modified: adding and removing a card are simple operations. On the other hand, an array allows constant access time to any card. Since our goal is to work on all the cards and not on some of them, each query accesses all the cards. Thus a list is a good choice. The same issue arises concerning the cards themselves: should they be lists or arrays of strings? This time an array is a good choice, since the format of a card is fixed for the whole database. It not possible to add a new field. Since a query might access only a few fields, it is important for this access to be fast.

The most natural solution for a card would be to use an array indexed by the names of the fields. Since such a type is not available in Objective Caml, we can use an array (indexed by integers) and a function associating a field name with the array index corresponding to the field.

```
# type data_card = string array ;;
# type data_base = { card_index : string → int ; data : data_card list } ;;
```

Access to the field named `n` of a card `dc` of the database `db` is implemented by the function:

```
# let field db n (dc : data_card) = dc.(db.card_index n) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

The type of `dc` has been set to *data_card* to constrain the function `field` to only accept string arrays and not arrays of other types.

Here is a small example:

```
# let base_ex =
    { data = [ [|"Chailloux"; "Emmanuel"|] ; [|"Manoury"; "Pascal"|] ]  ;
      card_index = function "Lastname"→0 | "Firstname"→1
                            | _->raise Not_found  } ;;
val base_ex : data_base =
  {card_index=<fun>;
   data=[[|"Chailloux"; "Emmanuel"|]; [|"Manoury"; "Pascal"|]]}
# List.map (field base_ex "Lastname") base_ex.data ;;
- : string list = ["Chailloux"; "Manoury"]
```

The expression *field base_ex* `"Lastname"` evaluates to a function which takes a card and returns the value of its `"Lastname"` field. The library function `List.map` applies the function to each card of the database `base_ex`, and returns the list of the results: a list of the `"Lastname"` fields of the database.

This example shows how we wish to use the functional style in our program. Here, the partial application of `field` allows us to define an access function for a given field, which we can use on any number of cards. This also shows us that the implementation of the `field` function is not very efficient, since although we are always accessing the same field, its index is computed for each access. The following implementation is better:

```
# let field base name =
    let i = base.card_index name in fun (card : data_card) → card.(i) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

Here, after applying the function to two arguments, the index of the field is computed and is used for any subsequent application.

Reading a database from a file

As seen from Objective Caml, a file containing a database is just a list of lines. The first work that needs to be done is to read each line as a string, split it into smaller parts according to the separating character, and then extract the corresponding data as well as the field indexing function.

Tools for processing a line

We need a function `split` that splits a string at every occurrence of some separating character. This function uses the function `suffix` which returns the suffix of a string `s` after some position `i`. To do this, we use three predefined functions:

- `String.length` returns the length of a string;
- `String.sub` returns the substring of `s` starting at position `i` and of length `l`;
- `String.index_from` computes the position of the first occurrence of character `c` in the string `s`, starting at position `n`.

```
# let suffix s i = try String.sub s i ((String.length s)-i)
                   with Invalid_argument("String.sub") → "" ;;
val suffix : string -> int -> string = <fun>
# let split c s =
    let rec split_from n =
      try  let p = String.index_from s n c
           in (String.sub s n (p-n)) :: (split_from (p+1))
      with Not_found → [ suffix s n ]
    in if s="" then [] else split_from 0 ;;
val split : char -> string -> string list = <fun>
```

The only remarkable characteristic in this implementation is the use of exceptions, specifically the exception `Not_found`.

Computing the *data_base* structure There is no difficulty in creating an array of strings from a list of strings, since this is what the `of_list` function in the `Array` module does. It might seem more complicated to compute the index function from a list of field names, but the `List` module provides all the needed tools.

Starting from a list of strings, we need to code a function that associates each string with an index corresponding to its position in the list.

```
# let mk_index list_names =
    let rec make_enum a b = if a > b then [] else a :: (make_enum (a+1) b) in
    let list_index =  (make_enum 0 ((List.length list_names) - 1)) in
    let assoc_index_name = List.combine list_names list_index  in
      function name → List.assoc name assoc_index_name  ;;
val mk_index : 'a list -> 'a -> int = <fun>
```

To create the association function between field names and indexes, we combine the list of indexes and the list of names to obtain a list of associations of the type *string * int list*. To look up the index associated with a name, we use the function `assoc` from the `List` library. The function `mk_index` returns a function that takes a name and calls `assoc` on this name and the previously built association list.

It is now possible to create a function that reads a file of the given format.

```
# let read_base filename =
    let channel = open_in filename in
    let split_line = split ':' in
    let list_names = split '|' (input_line channel) in
    let rec read_file () =
      try
        let data = Array.of_list (split_line (input_line channel )) in
          data :: (read_file ())
      with End_of_file →  close_in channel ; []
    in
      { card_index = mk_index list_names ; data = read_file () } ;;
val read_base : string -> data_base = <fun>
```

The auxiliary function `read_file` reads records from the file, and works recursively on the input channel. The base case of the recursion corresponds to the end of the file, signaled by the `End_of_file` exception. In this case, the empty list is returned after closing the channel.

The association's file can now be loaded:

```
# let base_ex = read_base "association.dat" ;;
val base_ex : data_base =
  {card_index=<fun>;
   data=
    [[|"0"; "Chailloux"; "Emmanuel"; "Universit\233 P6"; "0144274427";
       "ec@lip6.fr"; "email"; "25.12.1998"; "100.00"|];
     [|"1"; "Manoury"; "Pascal"; "Laboratoire PPS"; ...|]; ...]}
```

General principles for database processing

The effectiveness and difficulty of processing the data in a database is proportional to the power and complexity of the query language. Since we want to use Objective Caml as query language, there is no limit *a priori* on the requests we can express! However,

we also want to provide some simple tools to manipulate cards and their data. This desire for simplicity requires us to limit the power of the Objective Caml language, through the use of general goals and principles for database processing.

The goal of database processing is to obtain a *state* of the database. Building such a state may be decomposed into three steps:

1. selecting, according to some given criterion, a set of cards;
2. processing each of the selected cards;
3. processing all the data collected on the cards.

Figure 6.1 illustrates this decomposition.

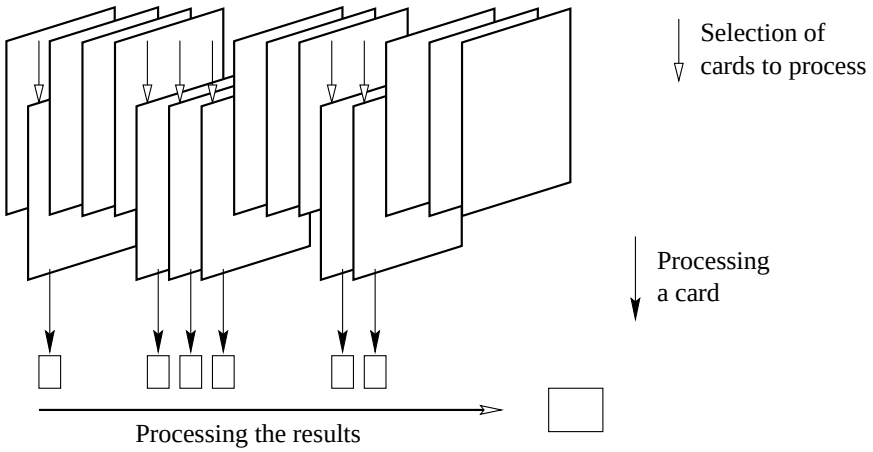

Figure 6.1: Processing a request.

According to this decomposition, we need three functions of the following types:

1. `(data_card -> bool) -> data_card list -> data_card list`
2. `(data_card -> 'a) -> data_card list -> 'a list`
3. `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

Objective Caml provides us with three higher-order function, also known as iterators, introduced page 219, that satisfy our specification:

```
# List.find_all ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

We will be able to use them to implement the three steps of building a state by choosing the functions they take as an argument.

For some special requests, we will also use:

```
# List.iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

Indeed, if the required processing consists only of displaying some data, there is nothing to compute.

In the next paragraphs, we are going to see how to define functions expressing simple selection criteria, as well as simple queries. We conclude this section with a short example using these functions according to the principles stated above.

Selection criteria

Concretely, the boolean function corresponding to the selection criterion of a card is a boolean combination of properties of some or all of the fields of the card. Each field of a card, even though it is a string, can contain some information of another type: a float, a date, etc.

Selection criteria on a field

Selecting on some field is usually done using a function of the type *data_base -> 'a -> string -> data_card -> bool*. The *'a* type parameter corresponds to the type of the information contained in the field. The *string* argument corresponds to the name of the field.

String fields We define two simple tests on strings: equality with another string, and non-emptiness.

```
# let eq_sfield db s n dc =  (s = (field db n dc)) ;;
val eq_sfield : data_base -> string -> string -> data_card -> bool = <fun>
# let nonempty_sfield db n dc = ("" <> (field db n dc)) ;;
val nonempty_sfield : data_base -> string -> data_card -> bool = <fun>
```

Float fields To implement tests on data of type float, it is enough to translate the *string* representation of a decimal number into its *float* value. Here are some examples obtained from a generic function `tst_ffield`:

```
# let tst_ffield r db v n dc = r v (float_of_string (field db n dc)) ;;
val tst_ffield :
  ('a -> float -> 'b) -> data_base -> 'a -> string -> data_card -> 'b = <fun>
# let eq_ffield = tst_ffield (=) ;;
# let lt_ffield = tst_ffield (<) ;;
# let le_ffield = tst_ffield (<=) ;;
(* etc. *)
```

These three functions have type:

data_base -> float -> string -> data_card -> bool.

Dates This kind of information is a little more complex to deal with, as it depends on the representation format of dates, and requires that we define date comparison.

We decide to represent dates in a card as a string with format `dd.mm.yyyy`. In order to be able to define additional comparisons, we also allow the replacement of the day, month or year part with the underscore character (`'_'`). Dates are compared according to the lexicographic order of lists of integers of the form *[year; month; day]*. To express queries such as: "is before July 1998", we use the *date pattern*: `"_.07.1998"`. Comparing a date with a pattern is accomplished with the function `tst_dfield` which analyses the pattern to create the *ad hoc* comparison function. To define this generic test function on dates, we need a few auxiliary functions.

We first code two conversion functions from dates (`ints_of_string`) and date patterns (`ints_of_dpat`) to lists of ints. The character `'_'` of a pattern will be replaced by the integer 0:

```
# let split_date = split '.' ;;
val split_date : string -> string list = <fun>
# let ints_of_string d =
    try  match split_date d with
            [d;m;y] → [int_of_string y; int_of_string m; int_of_string d]
          |    _   → failwith "Bad date format"
    with Failure("int_of_string") → failwith "Bad date format" ;;
val ints_of_string : string -> int list = <fun>

# let ints_of_dpat d =
    let int_of_stringpat = function "_" → 0 | s → int_of_string s
    in try  match split_date d with
                [d;m;y] → [ int_of_stringpat y; int_of_stringpat m;
                            int_of_stringpat d ]
              | _ → failwith "Bad date format"
       with Failure("int_of_string") → failwith "Bad date pattern" ;;
val ints_of_dpat : string -> int list = <fun>
```

Given a relation `r` on integers, we now code the test function. It simply consists of implementing the lexicographic order, taking into account the particular case of 0:

```
# let rec app_dtst r d1 d2 = match d1, d2 with
     [] , [] → false
   | (0 :: d1) , (_::d2) → app_dtst r d1 d2
   | (n1 :: d1) , (n2 :: d2) → (r n1 n2) || ((n1 = n2) && (app_dtst r d1 d2))
   | _, _ → failwith "Bad date pattern or format" ;;
val app_dtst : (int -> int -> bool) -> int list -> int list -> bool = <fun>
```

We finally define the generic function `tst_dfield` which takes as arguments a relation `r`, a database `db`, a pattern `dp`, a field name `nm`, and a card `dc`. This function checks that the pattern and the field from the card satisfy the relation.

```
# let tst_dfield r db dp nm dc =
```

```
    r (ints_of_dpat dp) (ints_of_string (field db nm dc)) ;;
val tst_dfield :
  (int list -> int list -> 'a) ->
  data_base -> string -> string -> data_card -> 'a = <fun>
```

We now apply it to three relations.

```
# let eq_dfield = tst_dfield (=) ;;
# let le_dfield = tst_dfield (<=) ;;
# let ge_dfield = tst_dfield (>=) ;;
```

These three functions have type:
data_base -> string -> string -> data_card -> bool.

Composing criteria

The tests we have defined above all take as first arguments a database, a value, and the name of a field. When we write a query, the value of these three arguments are known. For instance, when we work on the database `base_ex`, the test "is before July 1998" is written

```
# ge_dfield base_ex "_.07.1998" "Date" ;;
- : data_card -> bool = <fun>
```

Thus, we can consider a test as a function of type *data_card -> bool*. We want to obtain boolean combinations of the results of such functions applied to a given card. To this end, we implement the iterator:

```
# let fold_funs b c fs dc =
    List.fold_right (fun f → fun r → c (f dc) r) fs b ;;
val fold_funs : 'a -> ('b -> 'a -> 'a) -> ('c -> 'b) list -> 'c -> 'a = <fun>
```

Where `b` is the base value, the function `c` is the boolean operator, `fs` is the list of test functions on a field, and `dc` is a card.

We can obtain the conjunction and the disjunction of a list of tests with:

```
# let and_fold fs = fold_funs true (&) fs ;;
val and_fold : ('a -> bool) list -> 'a -> bool = <fun>
# let or_fold fs = fold_funs false (or) fs ;;
val or_fold : ('a -> bool) list -> 'a -> bool = <fun>
```

We easily define the negation of a test:

```
# let not_fun f dc = not (f dc) ;;
val not_fun : ('a -> bool) -> 'a -> bool = <fun>
```

For instance, we can use these combinators to define a selection function for cards whose date field is included in a given range:

```
# let date_interval db d1 d2 =
    and_fold [(le_dfield db d1 "Date"); (ge_dfield db d2 "Date")] ;;
val date_interval : data_base -> string -> string -> data_card -> bool =
```

```
  <fun>
```

Processing and computation

It is difficult to guess how a card might be processed, or the data that would result from that processing. Nevertheless, we can consider two common cases: numerical computation and data formatting for printing. Let's take an example for each of these two cases.

Data formatting

In order to print, we wish to create a string containing the name of a member of the association, followed by some information.

We start with a function that reverses the splitting of a line using a given separating character:

```
# let format_list c =
    let s = String.make 1 c in
    List.fold_left (fun x y → if x="" then y else x^s^y) "" ;;
val format_list : char -> string list -> string = <fun>
```

In order to build the list of fields we are interested in, we code the function `extract` that returns the fields associated with a given list of names in a given card:

```
# let extract db ns dc =
     List.map (fun n → field db n dc) ns  ;;
val extract : data_base -> string list -> data_card -> string list = <fun>
```

We can now write the line formatting function:

```
# let format_line db ns dc =
    (String.uppercase (field db "Lastname" dc))
    ^" "^(field db "Firstname" dc)
    ^"\t"^(format_list '\t' (extract db ns dc))
    ^"\n" ;;
val format_line : data_base -> string list -> data_card -> string = <fun>
```

The argument `ns` is the list of requested fields. In the resulting string, fields are separated by a tab (`'\t'`) and the string is terminated with a newline character.

We display the list of last and first names of all members with:

```
# List.iter print_string (List.map (format_line base_ex []) base_ex.data) ;;
CHAILLOUX Emmanuel
MANOURY Pascal
PAGANO Bruno
BARO Sylvain
- : unit = ()
```

Numerical computation

We want to compute the total amount of received fees for a given set of cards. This is easily done by composing the extraction and conversion of the correct field with the addition. To get nicer code, we define an infix composition operator:

```
# let (++) f g x = g (f x) ;;
val ++ : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

We use this operator in the following definition:

```
# let total db dcs =
    List.fold_right ((field db "Amount") ++ float_of_string ++ (+.)) dcs 0.0 ;;
val total : data_base -> data_card list -> float = <fun>
```

We can now apply it to the whole database:

```
# total base_ex base_ex.data ;;
- : float = 450
```

An example

To conclude, here is a small example of an application that uses the principles described in the paragraphs above.

We expect two kinds of queries on our database:

- a query returning two lists, the elements of the first containing the name of a member followed by his mail address, the elements of the other containing the name of the member followed by his email address, according to his preferences.
- another query returning the state of received fees for a given period of time. This state is composed of the list of last and first names, dates and amounts of the fees as well as the total amount of the received fees.

List of addresses

To create these lists, we first select the relevant cards according to the field `"Pref"`, then we use the formatting function `format_line`:

```
# let mail_addresses db =
   let dcs = List.find_all (eq_sfield db "mail" "Pref") db.data in
    List.map (format_line db ["Mail"]) dcs ;;
val mail_addresses : data_base -> string list = <fun>

# let email_addresses db =
   let dcs = List.find_all (eq_sfield db "email" "Pref") db.data in
    List.map (format_line db ["Email"]) dcs ;;
val email_addresses : data_base -> string list = <fun>
```

State of received fees

Computing the state of the received fees uses the same technique: selection then processing. In this case however the processing part is twofold: line formatting followed by the computation of the total amount.

```
# let fees_state db d1 d2 =
    let dcs = List.find_all (date_interval db d1 d2) db.data in
    let ls = List.map (format_line db ["Date";"Amount"]) dcs in
    let t = total db dcs in
     ls, t ;;
val fees_state : data_base -> string -> string -> string list * float = <fun>
```

The result of this query is a tuple containing a list of strings with member information, and the total amount of received fees.

Main program

The main program is essentially an interactive loop that displays the result of queries asked by the user through a menu. We use here an imperative style, except for the display of the results which uses an iterator.

```
# let main() =
    let db = read_base "association.dat" in
    let finished = ref false in
     while not !finished do
      print_string" 1: List of mail addresses\n";
      print_string" 2: List of email addresses\n";
      print_string" 3: Received fees\n";
      print_string" 0: Exit\n";
      print_string"Your choice: ";
      match read_int() with
        0 → finished := true
      | 1 → (List.iter print_string (mail_addresses db))
      | 2 → (List.iter print_string (email_addresses db))
      | 3
       → (let d1 = print_string"Start date: "; read_line() in
           let d2 = print_string"End date: "; read_line() in
           let ls, t = fees_state db d1 d2 in
            List.iter print_string ls;
            print_string"Total: "; print_float t; print_newline())
      | _ → ()
     done;
     print_string"bye\n" ;;
val main : unit -> unit = <fun>
```

This example will be extended in chapter 21 with an interface using a web browser.

Further work

A natural extension of this example would consist of adding type information to every field of the database. This information would be used to define generic comparison operators with type `data_base -> 'a -> string -> data_card -> bool` where the name of the field (the third argument) would trigger the correct conversion and test functions.

BASIC interpreter

The application described in this section is a program interpreter for Basic. Thus, it is a program that can run other programs written in Basic. Of course, we will only deal with a restricted language, which contains the following commands:

- **PRINT** *expression*
  Prints the result of the evaluation of the expression.
- **INPUT** *variable*
  Prints a *prompt* (`?`), reads an integer typed in by the user, and assigns its value to the variable.
- **LET** *variable* = *expression*
  Assigns the result of the evaluation of *expression* to the variable.
- **GOTO** *line number*
  Continues execution at the given line.
- **IF** *condition* **THEN** *line number*
  Continues execution at the given line if the *condition* is true.
- **REM** *any string*
  One-line comment.

Every line of a Basic program is labelled with a line number, and contains only one command. For instance, a program that computes and then prints the factorial of an integer given by the user is written:

```
 5  REM inputting the argument
10  PRINT " factorial of:"
20  INPUT A
30  LET B = 1
35  REM beginning of the loop
40  IF A <= 1 THEN 80
50  LET B = B * A
60  LET A = A - 1
70  GOTO 40
75  REM prints the result
80  PRINT B
```

We also wish to write a small text editor, working as a toplevel interactive loop. It should be able to add new lines, display a program, execute it, and display the result.

Execution of the program is started with the `RUN` command. Here is an example of the evaluation of this program:

```
> RUN
 factorial of: ? 5
120
```

The interpreter is implemented in several distinct parts:

Description of the abstract syntax : describes the definition of data types to represent Basic programs, as well as their components (lines, commands, expressions, etc.).

Program pretty printing : consists of transforming the internal representation of Basic programs to strings, in order to display them.

Lexing and parsing : accomplish the inverse transformation, that is, transform a string into the internal representation of a Basic program (the abstract syntax).

Evaluation : is the heart of the interpreter. It controls and runs the program. As we will see, functional languages, such as Objective Caml, are particularly well adapted for this kind of problem.

Toplevel interactive loop : glues together all the previous parts.

Abstract syntax

Figure 6.2 introduces the concrete syntax, as a BNF grammar, of the Basic we will implement. This kind of description for language syntaxes is described in chapter 11, page 295.

We can see that the way expressions are defined does not ensure that a *well formed* expression can be evaluated. For instance, `1+"hello"` is an expression, and yet it is not possible to evaluate it. This deliberate choice lets us simplify both the abstract syntax and the parsing of the Basic language. The price to pay for this choice is that a syntactically correct Basic program may generate a runtime error because of a type mismatch.

Defining Objective Caml data types for this abstract syntax is easy, we simply translate the concrete syntax into a sum type:

```
# type unr_op = UMINUS | NOT  ;;
# type bin_op = PLUS | MINUS | MULT | DIV | MOD
                | EQUAL | LESS | LESSEQ | GREAT | GREATEQ | DIFF
                | AND | OR  ;;
# type expression =
       ExpInt of int
     | ExpVar of string
     | ExpStr of string
```

UNARY_OP	::=	− | !
BINARY_OP	::=	+ | − | * | / | %
	|	= | < | > | <= | >= | <>
	|	& | '|'
EXPRESSION	::=	*integer*
	|	*variable*
	|	`"string"`
	|	UNARY_OP EXPRESSION
	|	EXPRESSION BINARY_OP EXPRESSION
	|	(EXPRESSION)
COMMAND	::=	**REM** *string*
	|	**GOTO** *integer*
	|	**LET** *variable* = EXPRESSION
	|	**PRINT** EXPRESSION
	|	**INPUT** *variable*
	|	**IF** EXPRESSION **THEN** *integer*
LINE	::=	*integer* COMMAND
PROGRAM	::=	LINE
	|	LINE PROGRAM
PHRASE	::=	LINE | **RUN** | **LIST** | **END**

Figure 6.2: BASIC Grammar.

```
    | ExpUnr of unr_op * expression
    | ExpBin of expression * bin_op * expression  ;;
# type command =
      Rem of string
    | Goto of int
    | Print of expression
    | Input of string
    | If of expression * int
    | Let of string * expression  ;;
# type line = { num : int ; cmd : command }  ;;
# type program = line list  ;;
```

We also define the abstract syntax for the commands for the small program editor:

```
# type phrase =  Line of line | List | Run | PEnd  ;;
```

It is convenient to allow the programmer to skip some parentheses in arithmetic expressions. For instance, the expression $1+3*4$ is usually interpreted as $1+(3*4)$. To this end, we associate an integer with each operator of the language:

```
# let priority_uop = function NOT → 1 | UMINUS → 7
  let priority_binop = function
      MULT | DIV  → 6
    | PLUS | MINUS → 5
    | MOD → 4
    | EQUAL | LESS | LESSEQ | GREAT | GREATEQ | DIFF → 3
    | AND | OR → 2 ;;
val priority_uop : unr_op -> int = <fun>
val priority_binop : bin_op -> int = <fun>
```

These integers indicate the *priority* of the operators. They will be used to print and parse programs.

Program pretty printing

To print a program, one needs to be able to convert abstract syntax program lines into strings.

Converting operators is easy:

```
# let pp_binop = function
      PLUS → "+" | MULT → "*" | MOD   → "%" | MINUS → "-"
    | DIV → "/"  | EQUAL → " = " | LESS → " < "
    | LESSEQ → " <= "   | GREAT   → " > "
    | GREATEQ → " >= "   | DIFF  → " <> " | AND → " & " | OR → " | "
  let pp_unrop = function UMINUS → "-"  | NOT → "!"  ;;
val pp_binop : bin_op -> string = <fun>
val pp_unrop : unr_op -> string = <fun>
```

Expression printing needs to take into account operator priority to print as few parentheses as possible. For instance, parentheses are put around a subexpression at the right of an operator only if the subexpression's main operator has a lower priority that the main operator of the whole expression. Also, arithmetic operators are left-associative, thus the expression $1-2-3$ is interpreted as $(1-2)-3$.

To deal with this, we use two auxiliary functions `ppl` and `ppr` to print left and right subtrees, respectively. These functions take two arguments: the tree to print and the priority of the enclosing operator, which is used to decide if parentheses are necessary. Left and right subtrees are distinguished to deal with associativity. If the current operator priority is the same than the enclosing operator priority, left trees do not need parentheses whereas right ones may require them, as in $1-(2-3)$ or $1-(2+3)$.

The initial tree is taken as a left subtree with minimal priority (0). The expression pretty printing function `pp_expression` is:

```
# let parenthesis x = "(" ^ x ^ ")";;
val parenthesis : string -> string = <fun>
# let pp_expression =
    let rec ppl pr = function
        ExpInt n → (string_of_int n)
      | ExpVar v → v
      | ExpStr s → "\"" ^ s ^ "\""
      | ExpUnr (op,e) →
          let res = (pp_unrop op)^(ppl (priority_uop op) e)
          in if pr=0 then res else parenthesis res
      | ExpBin (e1,op,e2) →
          let pr2 = priority_binop op
          in let res = (ppl pr2 e1)^(pp_binop op)^(ppr pr2 e2)
          (* parenthesis if priority is not greater *)
          in if pr2 >= pr then res else parenthesis res
    and ppr pr exp = match exp with
      (*  right subtrees only differ for binary operators *)
        ExpBin (e1,op,e2) →
          let pr2 = priority_binop op
          in let res = (ppl pr2 e1)^(pp_binop op)^(ppr pr2 e2)
          in if pr2 > pr then res else parenthesis res
      | _ → ppl pr exp
    in ppl 0 ;;
val pp_expression : expression -> string = <fun>
```

Command pretty printing uses the expression pretty printing function. Printing a line consists of printing the line number before the command.

```
# let pp_command = function
      Rem s     →  "REM " ^ s
    | Goto n    →  "GOTO " ^ (string_of_int n)
    | Print e   →  "PRINT " ^ (pp_expression e)
    | Input v   →  "INPUT " ^ v
    | If (e,n)  →  "IF "^(pp_expression e)^" THEN "^(string_of_int n)
    | Let (v,e) →  "LET " ^ v ^ " = " ^ (pp_expression e)  ;;
val pp_command : command -> string = <fun>
# let pp_line l = (string_of_int l.num) ^ "  " ^ (pp_command l.cmd)  ;;
val pp_line : line -> string = <fun>
```

Lexing

Lexing and parsing do the inverse transformation of printing, going from a string to a syntax tree. Lexing splits the text of a command line into independent lexical units called *lexemes*, with Objective Caml type:

```
# type lexeme = Lint of int
              | Lident of string
```

```
              | Lsymbol of string
              | Lstring of string
              | Lend ;;
```

A particular lexeme denotes the end of an expression: `Lend`. It is not present in the text of the expression, but is created by the lexing function (see the `lexer` function, page 165).

The string being lexed is kept in a record that contains a mutable field indicating the position after which lexing has not been done yet. Since the size of the string is used several times and does not change, it is also stored in the record:

```
# type string_lexer = {string:string; mutable current:int; size:int } ;;
```

This representation lets us define the lexing of a string as the application of a function to a value of type *string_lexer* returning a value of type *lexeme*. Modifying the current position in the string is done as a side effect.

```
# let init_lex s = { string=s; current=0 ; size=String.length s } ;;
val init_lex : string -> string_lexer = <fun>
# let forward cl = cl.current <- cl.current+1  ;;
val forward : string_lexer -> unit = <fun>
# let forward_n cl n = cl.current <- cl.current+n ;;
val forward_n : string_lexer -> int -> unit = <fun>
# let extract pred cl =
    let st = cl.string and pos = cl.current in
    let rec ext n = if n<cl.size && (pred st.[n]) then ext (n+1) else n in
    let res = ext pos
    in cl.current <- res ; String.sub cl.string pos (res-pos)  ;;
val extract : (char -> bool) -> string_lexer -> string = <fun>
```

The following functions extract a lexeme from the string and modify the current position. The two functions `extract_int` and `extract_ident` extract an integer and an identifier, respectively.

```
# let extract_int =
     let is_int = function '0'..'9' → true | _ → false
     in function cl → int_of_string (extract is_int cl)
  let extract_ident =
     let is_alpha_num = function
       'a'..'z' | 'A'..'Z' | '0' .. '9' | '_' → true
     | _ → false
     in extract is_alpha_num ;;
val extract_int : string_lexer -> int = <fun>
val extract_ident : string_lexer -> string = <fun>
```

The `lexer` function uses the two previous functions to extract a lexeme.

```
# exception LexerError ;;
exception LexerError
```

```
# let  rec lexer cl =
    let lexer_char c = match c with
        ' '
      | '\t'     → forward cl ; lexer cl
      | 'a'..'z'
      | 'A'..'Z' → Lident (extract_ident cl)
      | '0'..'9' → Lint (extract_int cl)
      | '"'      → forward cl ;
                   let res = Lstring (extract ((<>) '"') cl)
                   in forward cl ; res
      | '+' | '-' | '*' | '/' | '%' | '&' | '|' | '!' | '=' | '(' | ')'  →
                   forward cl; Lsymbol (String.make 1 c)
      | '<'
      | '>'      → forward cl;
                   if cl.current >= cl.size then Lsymbol (String.make 1 c)
                   else  let cs = cl.string.[cl.current]
                         in ( match (c,cs) with
                                ('<','=') → forward cl; Lsymbol "<="
                              | ('>','=') → forward cl; Lsymbol ">="
                              | ('<','>') → forward cl; Lsymbol "<>"
                              |     _     → Lsymbol (String.make 1 c) )
      | _ → raise LexerError
    in
       if cl.current >= cl.size then Lend
       else lexer_char cl.string.[cl.current] ;;
val lexer : string_lexer -> lexeme = <fun>
```

The `lexer` function is very simple: it matches the current character of a string and, based on its value, extracts the corresponding lexeme and modifies the current position to the start of the next lexeme. The code is simple because, for all characters except two, the current character defines which lexeme to extract. In the more complicated cases of `'<'`, we need to look at the next character, which might be a `'='` or a `'>'`, producing two different lexemes. The same problem arises with `'>'`.

Parsing

The only difficulty in parsing our language comes from expressions. Indeed, knowing the beginning of an expression is not enough to know its structure. For instance, having parsed the beginning of an expression as being $1 + 2 + 3$, the resulting syntax tree for this part depends on the rest of the expression: its structure is different when it is followed by $+4$ or $*4$ (see figure 6.3). However, since the tree structure for $1 + 2$ is the same in both cases, it can be built. As the position of $+3$ in the structure is not fully known, it is temporarily stored.

To build the abstract syntax tree, we use a *pushdown automaton* similar to the one built by *yacc* (see page 303). Lexemes are read one by one and put on a stack until

Figure 6.3: Basic: abstract syntax tree examples.

there is enough information to build the expression. They are then removed from the stack and replaced by the expression. This latter operation is called *reduction*.

The stack elements have type:

```
# type exp_elem =
      Texp of expression   (* expression       *)
    | Tbin of bin_op       (* binary operator  *)
    | Tunr of unr_op       (* unary operator   *)
    | Tlp                  (* left parenthesis *) ;;
```

Right parentheses are not stored on the stack as only left parentheses matter for reduction.

Figure 6.4 illustrates the way the stack is used to parse the expression $(1 + 2 * 3) + 4$. The character above the arrow is the current character of the string.

We define an exception for syntax errors.

```
# exception ParseError ;;
```

The first step consists of transforming symbols into operators:

```
# let unr_symb = function
    "!" → NOT | "-" → UMINUS | _ → raise ParseError
  let bin_symb = function
      "+" → PLUS | "-" → MINUS | "*" → MULT | "/" → DIV | "%" → MOD
    | "=" → EQUAL | "<" → LESS | "<=" → LESSEQ | ">" → GREAT
    | ">=" → GREATEQ | "<>" → DIFF | "&" → AND | "|" → OR
    | _ → raise ParseError
  let tsymb s = try Tbin (bin_symb s) with ParseError → Tunr (unr_symb s) ;;
val unr_symb : string -> unr_op = <fun>
val bin_symb : string -> bin_op = <fun>
val tsymb : string -> exp_elem = <fun>
```

The `reduce` function implements stack reduction. There are two cases to consider, whether the stack starts with:

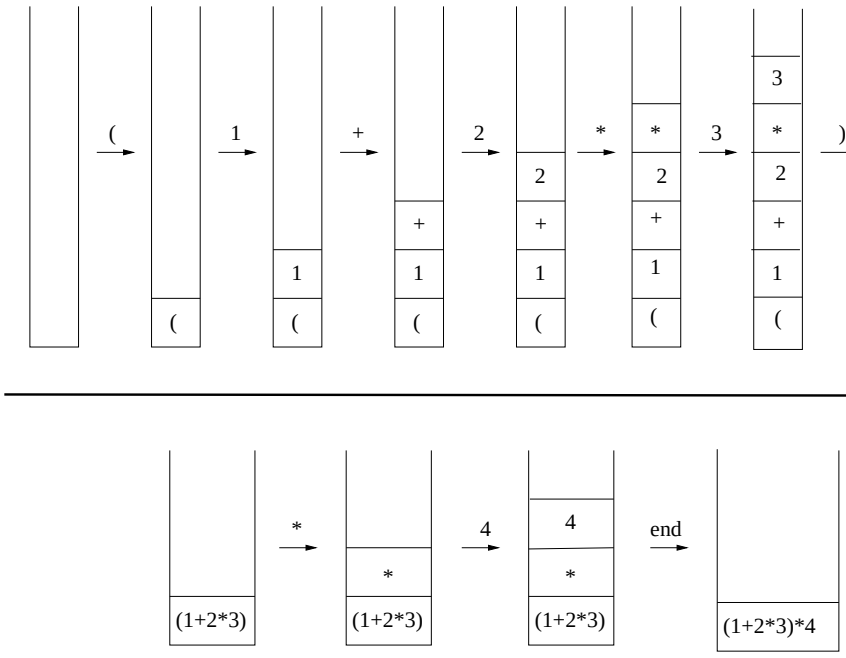

Figure 6.4: Basic: abstract syntax tree construction example.

- an expression followed by a unary operator,
- an expression followed by a binary operator and an expression.

Moreover, `reduce` takes an argument indicating the minimal priority that an operator should have to trigger reduction. To avoid this reduction condition, it suffices to give the minimal value, zero, as the priority.

```
# let reduce pr = function
      (Texp e) :: (Tunr op) :: st  when  (priority_uop op) >= pr
         → (Texp (ExpUnr (op,e))) :: st
    | (Texp e1) :: (Tbin op) :: (Texp e2) :: st  when  (priority_binop op) >= pr
         → (Texp (ExpBin (e2,op,e1))) :: st
    | _ → raise ParseError ;;
val reduce : int -> exp_elem list -> exp_elem list = <fun>
```

Notice that expression elements are stacked as they are read. Thus it is necessary to swap them when they are arguments of a binary operator.

The main function of our parser is `stack_or_reduce` that, according to the lexeme given in argument, puts it on the stack or triggers a reduction.

```
# let rec stack_or_reduce lex stack = match lex , stack with
      Lint n , _      →  (Texp (ExpInt n)) :: stack
    | Lident v , _    →  (Texp (ExpVar v)) :: stack
```

```
    | Lstring s , _    →  (Texp (ExpStr s)) :: stack
    | Lsymbol "(" , _  →  Tlp :: stack
    | Lsymbol ")" , (Texp e) :: Tlp :: st  →  (Texp e) :: st
    | Lsymbol ")" , _ → stack_or_reduce lex (reduce 0 stack)
    | Lsymbol s , _
        → let symbol =
            if s<>"-" then tsymb s
            (* remove the ambiguity of the ``-'' symbol           *)
            (* according to the last exp element put on the stack *)
            else match stack
                 with (Texp _)::_  →  Tbin MINUS
                               | _ →  Tunr UMINUS
          in ( match symbol with
                 Tunr op  →  (Tunr op) :: stack
               | Tbin op  →
                   ( try stack_or_reduce lex (reduce (priority_binop op)
                                                     stack )
                     with ParseError → (Tbin op) :: stack )
               | _ → raise ParseError )
    | _ , _ → raise ParseError ;;
val stack_or_reduce : lexeme -> exp_elem list -> exp_elem list = <fun>
```

Once all lexemes are defined and stacked, the function `reduce_all` builds the abstract syntax tree with the elements remaining in the stack. If the expression being parsed is well formed, only one element should remain in the stack, containing the tree for this expression.

```
# let rec reduce_all = function
    | [] → raise ParseError
    | [Texp x] → x
    | st → reduce_all (reduce 0 st) ;;
val reduce_all : exp_elem list -> expression = <fun>
```

The `parse_exp` function is the main expression parsing function. It reads a string, extracts its lexemes and passes them to the `stack_or_reduce` function. Parsing stops when the current lexeme satisfies a predicate that is given as an argument.

```
# let parse_exp stop cl =
    let p = ref 0 in
    let rec parse_one stack  =
      let l = ( p:=cl.current ; lexer cl)
      in if not (stop l) then parse_one (stack_or_reduce l stack)
         else ( cl.current <- !p ; reduce_all stack )
    in parse_one []  ;;
val parse_exp : (lexeme -> bool) -> string_lexer -> expression = <fun>
```

Notice that the lexeme that made the parsing stop is not used to build the expression. It is thus necessary to modify the current position to its beginning (variable `p`) to parse it later.

We can now parse a command line:

```
# let parse_cmd cl = match lexer cl with
      Lident s → ( match s with
          "REM" → Rem (extract (fun _ → true) cl)
        | "GOTO" → Goto (match lexer cl with
                           Lint p → p
                         | _ → raise ParseError)
        | "INPUT" → Input (match lexer cl with
                             Lident v → v
                           | _ → raise ParseError)
        | "PRINT" → Print (parse_exp ((=) Lend) cl)
        | "LET" →
            let l2 = lexer cl and l3 = lexer cl
            in ( match l2 ,l3 with
                     (Lident v,Lsymbol "=") → Let (v,parse_exp ((=) Lend) cl)
                   | _ → raise ParseError )
        | "IF" →
            let test = parse_exp ((=) (Lident "THEN")) cl
            in ( match ignore (lexer cl) ; lexer cl with
                    Lint n  →  If (test,n)
                  | _ → raise ParseError )
        | _ → raise ParseError )
    | _ → raise ParseError  ;;
val parse_cmd : string_lexer -> command = <fun>
```

Finally, we implement the function to parse commands typed by the user:

```
# let parse str =
    let cl = init_lex str
    in match lexer cl with
        Lint n → Line { num=n ; cmd=parse_cmd cl }
      | Lident "LIST" → List
      | Lident "RUN" → Run
      | Lident "END" → PEnd
      | _ → raise ParseError  ;;
val parse : string -> phrase = <fun>
```

Evaluation

A Basic program is a list of lines. Execution starts at the first line. Interpreting a program line consists of executing the task corresponding to its command. There are three different kinds of commands: input-output (**PRINT** and **INPUT**), variable declaration or modification (**LET**), and flow control (**GOTO** and **IF... THEN**). Input-output commands interact with the user and use the corresponding Objective Caml functions.

Variable declaration and modification commands need to know how to compute the value of an arithmetic expression and the memory location to store the result. Expression evaluation returns an integer, a boolean, or a string. Their type is `value`.

```
# type value = Vint of int | Vstr of string | Vbool of bool  ;;
```

Variable declaration should allocate some memory to store the associated value. Similarly, variable modification requires the modification of the associated value. Thus, evaluation of a Basic program uses an *environment* that stores the association between a variable name and its value. It is represented by an association list of tuples (name,value):

```
# type environment = (string * value) list ;;
```

The variable name is used to access its value. Variable modification modifies the association.

Flow control commands, conditional or unconditional, specify the number of the next line to execute. By default, it is the next line. To do this, it is necessary to remember the number of the current line.

The list of commands representing the program being edited under the toplevel is not an efficient data structure for running the program. Indeed, it is then necessary to look at the whole list of lines to find the line indicated by a flow control command (`If` and `goto`). Replacing the list of lines with an array of commands allows direct access to the command following a flow control command, using the array index instead of the line number in the flow control command. This solution requires some preprocessing called *assembly* before executing a `RUN` command. For reasons that will be detailed shortly, a program after assembly is not represented as an array of commands but as an array of lines:

```
# type code = line array ;;
```

As in the calculator example of previous chapters, the interpreter uses a state that is modified for each command evaluation. At each step, we need to remember the whole program, the next line to interpret and the values of the variables. The program being interpreted is not exactly the one that was entered in the toplevel: instead of a list of commands, it is an array of commands. Thus the state of a program during execution is:

```
# type state_exec = { line:int ; xprog:code ; xenv:environment } ;;
```

Two different reasons may lead to an error during the evaluation of a line: an error while computing an expression, or branching to an absent line. They must be dealt with so that the interpreter exits nicely, printing an error message. We define an exception as well as a function to raise it, indicating the line where the error occurred.

```
# exception RunError of int
  let runerr n = raise (RunError n) ;;
exception RunError of int
val runerr : int -> 'a = <fun>
```

Assembly Assembling a program that is a list of numbered lines (type *program*) consists of transforming this list into an array and modifying the flow control commands. This last modification only needs an association table between line numbers and array indexes. This is easily provided by storing lines (with their line numbers), instead of commands, in the array: to find the association between a line number and the index in the array, we look the line number up in the array and return the corresponding index. If no line is found with this number, the index returned is `-1`.

```
# exception Result_lookup_index of int ;;
exception Result_lookup_index of int
# let lookup_index tprog num_line =
    try
      for i=0 to (Array.length tprog)-1 do
        let num_i = tprog.(i).num
        in if num_i=num_line then raise (Result_lookup_index i)
           else if num_i>num_line then raise (Result_lookup_index (-1))
      done ;
      (-1 )
    with Result_lookup_index i → i ;;
val lookup_index : line array -> int -> int = <fun>

# let assemble prog =
    let tprog = Array.of_list prog in
      for i=0 to (Array.length tprog)-1 do
        match tprog.(i).cmd with
            Goto n → let index = lookup_index tprog n
                     in tprog.(i) <- { tprog.(i) with cmd = Goto index }
          | If(c,n) → let index = lookup_index tprog n
                      in tprog.(i) <- { tprog.(i) with cmd = If (c,index) }
          | _ → ()
      done ;
    tprog ;;
val assemble : line list -> line array = <fun>
```

Expression evaluation The evaluation function does a depth-first traversal on the abstract syntax tree, and executes the operations indicated at each node.

The `RunError` exception is raised in case of type inconsistency, division by zero, or an undeclared variable.

```
# let rec eval_exp n envt expr = match expr with
      ExpInt p  →  Vint p
    | ExpVar v  → ( try List.assoc v envt with Not_found → runerr n )
    | ExpUnr (UMINUS,e) →
        ( match eval_exp n envt e with
            Vint p → Vint (-p)
          | _ → runerr n )
    | ExpUnr (NOT,e) →
```

```
        ( match eval_exp n envt e with
            Vbool p → Vbool (not p)
          | _ → runerr n )
    | ExpStr s → Vstr s
    | ExpBin (e1,op,e2)
        → match eval_exp n envt e1 , op , eval_exp n envt e2 with
                Vint v1 , PLUS   , Vint v2  →  Vint (v1 + v2)
              | Vint v1 , MINUS  , Vint v2  →  Vint (v1 - v2)
              | Vint v1 , MULT   , Vint v2  →  Vint (v1 * v2)
              | Vint v1 ,  DIV   , Vint v2  when v2<>0 →  Vint (v1 / v2)
              | Vint v1 ,  MOD   , Vint v2  when v2<>0 →  Vint (v1 mod v2)

              | Vint v1 , EQUAL   , Vint v2  →  Vbool (v1 = v2)
              | Vint v1 , DIFF    , Vint v2  →  Vbool (v1 <> v2)
              | Vint v1 ,  LESS   , Vint v2  →  Vbool (v1 < v2)
              | Vint v1 ,  GREAT  , Vint v2  →  Vbool (v1 > v2)
              | Vint v1 , LESSEQ  , Vint v2  →  Vbool (v1 <= v2)
              | Vint v1 , GREATEQ , Vint v2  →  Vbool (v1 >= v2)

              | Vbool v1 , AND , Vbool v2  →  Vbool (v1 && v2)
              | Vbool v1 , OR , Vbool v2  →  Vbool (v1 || v2)

              | Vstr v1 , PLUS , Vstr v2 → Vstr (v1 ^ v2)
              | _ , _ , _  → runerr n  ;;
val eval_exp : int -> (string * value) list -> expression -> value = <fun>
```

Command evaluation To evaluate a command, we need a few additional functions.

We add an association to an environment by removing a previous association for the same variable name if there is one:

```
# let rec add v e env = match env with
      []  → [v,e]
    | (w,f) :: l → if w=v then (v,e) :: l else (w,f) :: (add v e l) ;;
val add : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

A function that prints the value of an integer or string is useful for evaluation of the PRINT command.

```
# let print_value v = match v with
      Vint n → print_int n
    | Vbool true → print_string "true"
    | Vbool false → print_string "false"
    | Vstr s → print_string s ;;
val print_value : value -> unit = <fun>
```

The execution of a command corresponds to a *transition* from one state to another. More precisely, the environment is modified if the command is an assignment. Furthermore, the next line to execute is always modified. As a convention, if the next line to execute does not exist, we set its value to `-1`

```
# let next_line state =
    let n = state.line+1 in
     if n < Array.length state.xprog then n else -1 ;;
val next_line : state_exec -> int = <fun>
# let eval_cmd state =
     match state.xprog.(state.line).cmd with
       Rem _     →  { state with line = next_line state }
     | Print e   →  print_value (eval_exp state.line state.xenv e) ;
                    print_newline () ;
                    { state with line = next_line state }
     | Let(v,e)  →  let ev = eval_exp state.line state.xenv e
                      in { state with line = next_line state ;
                                      xenv = add v ev state.xenv }
     | Goto n    →  { state with line = n }
     | Input v   →  let x = try read_int ()
                            with Failure "int_of_string" → 0
                    in { state with line = next_line state;
                                    xenv = add v (Vint x) state.xenv }
     | If (t,n)  →  match eval_exp state.line state.xenv t with
                      Vbool true  → { state with line = n }
                    | Vbool false → { state with line = next_line state }
                    | _ → runerr state.line  ;;
val eval_cmd : state_exec -> state_exec = <fun>
```

On each call of the transition function `eval_cmd`, we look up the current line, run it, then set the number of the next line to run as the current line. If the last line of the program is reached, the current line is given the value `-1`. This will tell us when to stop.

Program evaluation We recursively apply the transition function until we reach a state where the current line number is `-1`.

```
# let rec run state =
    if state.line = -1 then state else run (eval_cmd state) ;;
val run : state_exec -> state_exec = <fun>
```

Finishing touches

The only thing left to do is to write a small editor and to plug together all the functions we wrote in the previous sections.

The `insert` function adds a new line in the program at the requested place.

```
# let rec insert line p = match p with
       [] → [line]
    | l :: prog →
        if l.num < line.num then l :: (insert line prog)
        else if l.num=line.num then line :: prog
        else line :: l :: prog ;;
val insert : line -> line list -> line list = <fun>
```

The `print_prog` function prints the source code of a program.

```
# let print_prog prog =
    let print_line x = print_string (pp_line x) ; print_newline () in
     print_newline () ;
     List.iter print_line prog ;
     print_newline () ;;
val print_prog : line list -> unit = <fun>
```

The `one_command` function processes the insertion of a line or the execution of a command. It modifies the state of the toplevel loop, which consists of a program and an environment. This state, represented by the *loop_state* type, is different from the evaluation state.

```
# type loop_state = { prog:program; env:environment } ;;
# exception End ;;

# let one_command state =
    print_string "> " ; flush stdout ;
    try
      match parse (input_line stdin) with
         Line l → { state with prog = insert l state.prog }
       | List  → (print_prog state.prog ; state )
       | Run
        → let tprog = assemble state.prog in
           let xstate = run { line = 0; xprog = tprog; xenv = state.env } in
            {state with env = xstate.xenv }
       | PEnd → raise End
    with
        LexerError → print_string "Illegal character\n"; state
      | ParseError → print_string "syntax error\n"; state
      | RunError n →
          print_string "runtime error at line ";
          print_int n ;
          print_string "\n";
          state ;;
val one_command : loop_state -> loop_state = <fun>
```

The main function is the `go` function, which starts the toplevel loop of our Basic.

```
# let go () =
  try
    print_string "Mini-BASIC version 0.1\n\n";
    let rec loop state = loop (one_command state)  in
      loop { prog = []; env = [] }
  with End → print_string "See you later...\n";;
val go : unit -> unit = <fun>
```

The loop is implemented by the local function `loop`. It stops when the `End` exception is raised by the `one_command` function.

Example: C+/C-

We return to the example of the C+/C- game described in chapter 3, page 78. Here is the Basic program corresponding to that Objective Caml program:

```
10 PRINT "Give the hidden number: "
20 INPUT N
30 PRINT "Give a number: "
40 INPUT R
50 IF R = N THEN 110
60 IF R < N THEN 90
70 PRINT "C-"
80 GOTO 30
90 PRINT "C+"
100 GOTO 30
110 PRINT "CONGRATULATIONS"
```

And here is a sample run of this program.

```
> RUN
Give the hidden number:
64
Give a number:
88
C-
Give a number:
44
C+
Give a number:
64
CONGRATULATIONS
```

Further work

The Basic we implemented is minimalist. If you want to go further, the following exercises hint at some possible extensions.

1. *Floating-point numbers*: as is, our language only deals with integers, strings and booleans. Add floats, as well as the corresponding arithmetic operations in the language grammar. We need to modify not only parsing, but also evaluation, taking into account the implicit conversions between integers and floats.
2. *Arrays*: Add to the syntax the command `DIM var[x]` that declares an array `var` of size `x`, and the expression `var[i]` that references the `i`th element of the array `var`.
3. *Toplevel directives*: Add the toplevel directives `SAVE "file_name"` and `LOAD "file_name"` that save a Basic program to the hard disk, and load a Basic program from the hard disk, respectively.
4. *Sub-program*: Add sub-programs. The `GOSUB line number` command calls a sub-program by branching to the given line number while storing the line from where the call is made. The `RETURN` command resumes execution at the line following the last `GOSUB` call executed, if there is one, or exits the program otherwise. Adding sub-programs requires evaluation to manage not only the environement but also a stack containing the return addresses of the current `GOSUB` calls. The `GOSUB` command adds the possibility of defining recursive sub-programs.

Minesweeper

Let us briefly recall the object of this game: to explore a mine field without stepping on one. A mine field is a two dimensional array (a matrix) where some cells contain hidden mines while others are empty. At the beginning of the game, all the cells are closed and the player must open them one after another. The player wins when he opens all the cells that are empty.

Every turn, the player may open a cell or flag it as containing a mine. If he opens a cell that contains a mine, it blows up and the player loses. If the cell is empty, its appearance is modified and the number of mines in the 8 neighbor cells is displayed (thus at most 8). If the player decides to flag a cell, he cannot open it until he removes the flag.

We split the implementation of the game into three parts.

1. The abstract game, including the internal representation of the mine field as well as the functions manipulating this representation.
2. The graphical part of the game, including the function for displaying cells.
3. The interaction between the program and the player.

Figure 6.5: Screenshot.

The abstract mine field

This part deals with the mine field as an abstraction only, and does not address its display.

Configuration A mine field is defined by its dimensions and the number of mines it contains. We group these three pieces of data in a record and define a default configuration: 10×10 cells and 15 mines.

```
# type config = {
    nbcols  : int ;
    nbrows : int ;
    nbmines : int };;
# let default_config = { nbcols=10; nbrows=10; nbmines=15 } ;;
```

The mine field It is natural to represent the mine field as a two dimensional array. However, it is still necessary to specify what the cells are, and what information their encoding should provide. The state of a cell should answer the following questions:

- is there a mine in this cell?
- is this cell opened (has it been seen)?
- is this cell flagged?
- how many mines are there in neighbor cells?

The last item is not mandatory, as it is possible to compute it when it is needed. However, it is simpler to do this computation once at the beginning of the game.

We represent a cell with a record that contains these four pieces of data.

```
# type cell = {
    mutable mined : bool ;
    mutable seen : bool ;
    mutable flag : bool ;
    mutable nbm : int
  } ;;
```

The two dimensional array is an array of arrays of cells:

```
# type board = cell array array  ;;
```

An iterator In the rest of the program, we often need to iterate a function over all the cells of the mine field. To do it generically, we define the operator `iter_cells` that applies the function `f`, given as an argument, to each cell of the board defined by the configuration `cf`.

```
# let iter_cells cf f =
    for i=0 to cf.nbcols-1 do for j=0 to cf.nbrows-1 do f (i,j) done done ;;
val iter_cells : config -> (int * int -> 'a) -> unit = <fun>
```

This is a good example of a mix between functional and imperative programming styles, as we use a higher order function (a function taking another function as an argument) to iterate a function that operates through side effects (as it returns no value).

Initialization We randomly choose which cells are mines. If c and r are respectively the number of columns and rows of the mine field, and m the number of mines, we need to generate m different numbers between 1 and $c \times r$. We suppose that $m \leq c \times r$ to define the algorithm, but the program using it will need to check this condition.

The straightforward algorithm consists of starting with an empty list, picking a random number and putting it in the list if it is not there already, and repeating this until the list contains m numbers. We use the following functions from the `Random` and `Sys` modules:

- `Random.int`: *int -> int*, picks a number between 0 and $n-1$ (n is the argument) according to a random number generator;
- `Random.init`: *int -> unit*, initializes the random number generator;

- `Sys.time`: *`unit -> float`*, returns the number of milliseconds of processor time the program used since it started. This function will be used to initialize the random number generator with a different seed for each game.

The modules containing these functions are described in more details in chapter 8, pages 216 and 234.

The random mine placement function receives the number of cells (`cr`) and the number of mines to place (`m`), and returns a list of linear positions for the `m` mines.

```
# let random_list_mines cr m =
    let cell_list = ref []
    in while (List.length !cell_list) < m do
         let n = Random.int cr in
          if not (List.mem n !cell_list) then cell_list := n :: !cell_list
       done ;
      !cell_list ;;
val random_list_mines : int -> int -> int list = <fun>
```

With such an implementation, there is no upper bound on the number of steps the function takes to terminate. If the random number generator is reliable, we can only insure that the probability it does not terminate is zero. However, all experimental uses of this function have never failed to terminate. Thus, even though it is not guaranteed that it will terminate, we will use it to generate the list of mined cells.

We need to initialize the random number generator so that each run of the game does not use the same mine field. We use the processor time since the beginning of the program execution to initialize the random number generator.

```
# let generate_seed () =
    let t = Sys.time () in
    let n = int_of_float (t*.1000.0)
    in Random.init(n mod 100000) ;;
val generate_seed : unit -> unit = <fun>
```

In practice, a given program very often takes the same execution time, which results in a similar result for `generate_seed` for each run. We ought to use the `Unix.time` function (see chapter 18).

We very often need to know the neighbors of a given cell, during the initialization of the mine field as well as during the game. Thus we write a `neighbors` function. This function must take into account the side and corner cells that have fewer neighbors than the middle ones (function `valid`).

```
# let valid cf (i,j) = i>=0 && i<cf.nbcols && j>=0 && j<cf.nbrows  ;;
val valid : config -> int * int -> bool = <fun>
# let neighbors cf (x,y) =
    let ngb = [x-1,y-1; x-1,y; x-1,y+1; x,y-1; x,y+1; x+1,y-1; x+1,y; x+1,y+1]
    in List.filter (valid cf) ngb ;;
val neighbors : config -> int * int -> (int * int) list = <fun>
```

The `initialize_board` function creates the initial mine field. It proceeds in four steps:

1. generation of the list of mined cells;
2. creation of a two dimensional array containing different cells;
3. setting of mined cells in the board;
4. computation of the number of mines in neighbor cells for each cell that is not mined.

The function `initialize_board` uses a few local functions that we briefly describe.

`cell_init` : creates an initial cell value;

`copy_cell_init` : puts a copy of the initial cell value in a cell of the board;

`set_mined` : puts a mine in a cell;

`count_mined_adj` : computes the number of mines in the neighbors of a given cell;

`set_count` : updates the number of mines in the neighbors of a cell if it is not mined.

```
# let initialize_board cf =
    let cell_init () = { mined=false; seen=false; flag=false; nbm=0 } in
    let copy_cell_init b (i,j) = b.(i).(j) <- cell_init() in
    let set_mined b n = b.(n / cf.nbrows).(n mod cf.nbrows).mined <- true
    in
    let count_mined_adj b (i,j) =
      let x = ref 0 in
      let inc_if_mined (i,j) = if b.(i).(j).mined then incr x
      in List.iter inc_if_mined (neighbors cf (i,j)) ;
         !x
    in
    let set_count b (i,j) =
      if not b.(i).(j).mined
      then b.(i).(j).nbm <- count_mined_adj b (i,j)
    in
    let list_mined = random_list_mines (cf.nbcols*cf.nbrows) cf.nbmines in
    let board = Array.make_matrix cf.nbcols cf.nbrows (cell_init ())
    in iter_cells cf (copy_cell_init board) ;
       List.iter (set_mined board) list_mined ;
       iter_cells cf (set_count board) ;
       board ;;
val initialize_board : config -> cell array array = <fun>
```

Opening a cell During a game, when the player opens a cell whose neighbors are empty (none contains a mine), he knows that he can open the neighboring cells without risk, and he can keep opening cells as long as he opens cells without any mined neighbor. In order to relieve the player of this boring process (as it is not challenging at all), our

Minesweeper opens all these cells itself. To this end, we write the function `cells_to_see` that returns a list of all the cells to open when a given cell is opened.

The algorithm needed is simple to state: if the opened cell has some neighbors that contain a mine, then the list of cells to see consists only of the opened cell; otherwise, the list of cells to see consists of the neighbors of the opened cell, as well as the lists of cells to see of these neighbors. The difficulty is in writing a program that does not loop, as every cell is a neighbor of any of its neighbors. We thus need to avoid processing the same cell twice.
To remember which cells were processed, we use the array of booleans `visited`. Its size is the same as the mine field. The value **`true`** for a cell of this array denotes that it was already visited. We recurse only on cells that were not visited.

We use the auxiliary function `relevant` that computes two sublists from the list of neighbors of a cell. Each one of these lists only contains cells that do not contain a mine, that are not opened, that are not flagged by the player, and that were not visited. The first sublist is the list of neighboring cells who have at least one neighbor containing a mine; the second sublist is the list of neighboring cells whose neighbors are all empty. As these lists are computed, all these cells are marked as visited. Notice that flagged cells are not processed, as a flag is meant to prevent opening a cell.

The local function `cells_to_see_rec` implements the recursive search loop. It takes as an argument the list of cells to visit, updates it, and returns the list of cells to open. This function is called with the list consisting only of the cell being opened, after it is marked as visited.

```
# let cells_to_see bd cf (i,j) =
    let visited = Array.make_matrix cf.nbcols cf.nbrows false in
    let rec relevant = function
        [] → ([],[])
      | ((x,y) as c) :: t →
          let cell=bd.(x).(y)
          in if cell.mined || cell.flag || cell.seen || visited.(x).(y)
             then relevant t
             else let (l1,l2) = relevant t
                  in visited.(x).(y) <- true ;
                     if cell.nbm=0 then (l1,c :: l2) else (c :: l1,l2)
    in
    let rec cells_to_see_rec = function
        [] → []
      | ((x,y) as c) :: t →
          if bd.(x).(y).nbm<>0 then c :: (cells_to_see_rec t)
          else let (l1,l2) = relevant (neighbors cf c)
               in (c :: l1) @ (cells_to_see_rec (l2 @ t))
    in visited.(i).(j) <- true ;
       cells_to_see_rec [(i,j)] ;;
val cells_to_see :
  cell array array -> config -> int * int -> (int * int) list = <fun>
```

At first sight, the argument of `cells_to_see_rec` may grow between two consecutive calls, although the recursion is based on this argument. It is legitimate to wonder if this function always terminates.
The way the `visited` array is used guarantees that a visited cell cannot be in the result of the `relevant` function. Also, all the cells to visit come from the result of the `relevant` function. As the `relevant` function marks as visited all the cells it returns, it returns each cell at most once, thus a cell may be added to the list of cells to visit at most once. The number of cells being finite, we deduce that the function terminates.

Except for graphics, we are done with our Minesweeper. Let us take a look at the programming style we have used. Mutable structures (arrays and mutable record fields) make us use an imperative style of loops and assignments. However, to deal with auxiliary issues, we use lists that are processed by functions written in a functional style. Actually, the programming style is a consequence of the data structure that it manipulates. The function `cells_to_see` is a good example: it processes lists, and it is natural to write it in a functional style. Nevertheless, we use an array to remember the cells that were already processed, and we update this array imperatively. We could use a purely functional style by using a list of visited cells instead of an array, and check if a cell is in the list to see if it was visited. However, the cost of such a choice is important (looking up an element in a list is linear in the size of the list, whereas accessing an array element takes constant time) and it does not make the program simpler.

Displaying the Minesweeper game

This part depends on the data structures representing the state of the game (see page 177). It consists of displaying the different components of the Minesweeper window, as shown in figure 6.6. To this end, we use the box drawing functions seen on page 126.

The following parameters characterize the components of the graphical window.

```
# let b0 = 3 ;;
# let w1 = 15 ;;
# let w2 = w1 ;;
# let w4 = 20 + 2*b0 ;;
# let w3 = w4*default_config.nbcols + 2*b0 ;;
# let w5 = 40 + 2*b0 ;;
```

```
# let h1 = w1 ;;
# let h2 = 30 ;;
# let h3 = w5+20 + 2*b0 ;;
# let h4 = h2 ;;
# let h5 = 20 + 2*b0 ;;
# let h6 = w5 + 2*b0 ;;
```

We use them to extend the basic configuration of our Minesweeper board (value of type *config*). Below, we define a record type *window_config*. The `cf` field contains the basic configuration. We associate a box with every component of the display: main window (field `main_box`), mine field (field `field_box`), dialog window (field `dialog_box`) with two sub-boxes (fields `d1_box` and `d2_box`), flagging button (field `flag_box`) and current cell (field `current_box`).

```
# type window_config = {
    cf : config ;
```

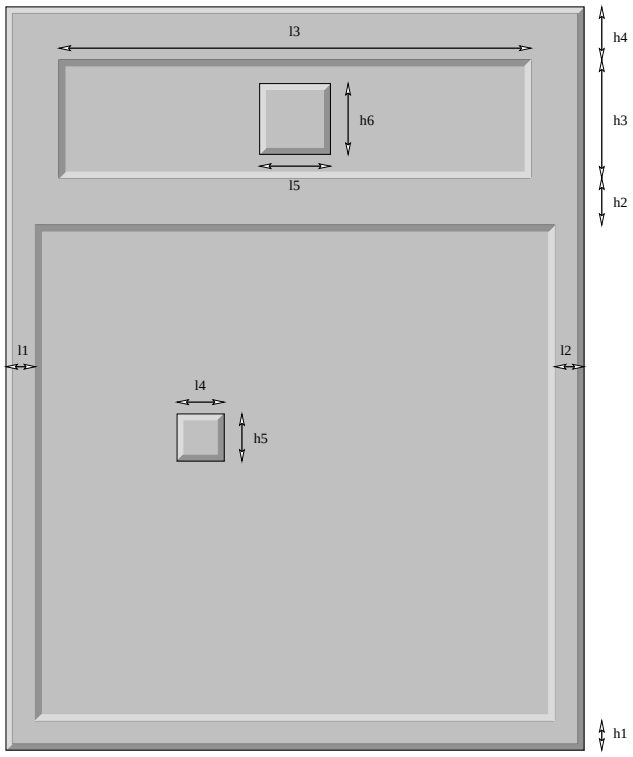

Figure 6.6: The main window of Minesweeper.

```
    main_box : box_config ;
    field_box : box_config ;
    dialog_box : box_config ;
    d1_box : box_config ;
    d2_box : box_config ;
    flag_box : box_config ;
    mutable current_box : box_config ;
    cell : int*int → (int*int) ;
    coor : int*int → (int*int)
  } ;;
```

Moreover, a record of type *window_config* contains two functions:

- `cell`: takes the coordinates of a cell and returns the coordinates of the corresponding box;
- `coor`: takes the coordinates of a pixel of the window and returns the coordinates of the corresponding cell.

Configuration We now define a function that builds a graphical configuration (of type *window_config*) according to a basic configuration (of type *config*) and the parameters above. The values of the parameters of some components depend on the value of the parameters of other components. For instance, the global box width depends on the mine field width, which, in turn, depends on the number of columns. To avoid computing the same value several times, we incrementally create the components. This initialization phase of a graphical configuration is always a little tedious when there is no adequate primitive or tool available.

```
# let make_box x y w h bw r =
    { x=x; y=y; w=w; h=h; bw=bw; r=r; b1_col=gray1; b2_col=gray3; b_col=gray2 } ;;
val make_box : int -> int -> int -> int -> int -> relief -> box_config =
  <fun>
# let make_wcf cf =
    let wcols =  b0 + cf.nbcols*w4 + b0
    and hrows =  b0 + cf.nbrows*h5 + b0  in
    let main_box =  let gw = (b0 + w1 + wcols + w2 + b0)
                    and gh = (b0 + h1 + hrows + h2 + h3 + h4 + b0)
                    in make_box 0 0 gw gh b0 Top
    and field_box = make_box w1 h1 wcols hrows b0 Bot  in
    let dialog_box = make_box  ((main_box.w - w3) / 2)
                               (b0+h1+hrows+h2)
                               w3 h3 b0 Bot
    in
    let d1_box = make_box (dialog_box.x + b0) (b0 + h1 + hrows + h2)
                          ((w3-w5)/2-(2*b0)) (h3-(2*b0)) 5 Flat  in
    let flag_box = make_box (d1_box.x + d1_box.w)
                         (d1_box.y + (h3-h6) / 2) w5 h6 b0 Top  in
    let d2_box = make_box (flag_box.x + flag_box.w)
                          d1_box.y d1_box.w d1_box.h 5 Flat  in
    let current_box = make_box 0 0 w4 h5 b0 Top
    in { cf = cf;
         main_box = main_box; field_box=field_box; dialog_box=dialog_box;
         d1_box=d1_box;
         flag_box=flag_box; d2_box=d2_box; current_box = current_box;
         cell = (fun (i,j) → ( w1+b0+w4*i , h1+b0+h5*j)) ;
         coor = (fun (x,y) → ( (x-w1)/w4 , (y-h1)/h5 )) } ;;
val make_wcf : config -> window_config = <fun>
```

Cell display We now need to write the functions to display the cells in their different states. A cell may be open or closed and may contain some information. We always display (the box corresponding with) the current cell in the game configuration (field `cc_bcf`).

We thus write two functions modifying the configuration of the current cell; one closing it, the other opening it.

```
# let close_ccell wcf i j =
```

```
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Top} ;;
val close_ccell : window_config -> int -> int -> unit = <fun>
# let open_ccell wcf i j =
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Flat} ;;
val open_ccell : window_config -> int -> int -> unit = <fun>
```

Depending on the game phase, we may need to display some information on the cells. We write, for each case, a specialized function.

- Display of a closed cell:

```
# let draw_closed_cc wcf i j =
    close_ccell wcf i j;
    draw_box wcf.current_box ;;
val draw_closed_cc : window_config -> int -> int -> unit = <fun>
```

- Display of an opened cell with its number of neighbor mines:

```
# let draw_num_cc wcf i j n =
     open_ccell wcf i j ;
     draw_box wcf.current_box ;
     if n<>0 then draw_string_in_box Center (string_of_int n)
                                        wcf.current_box Graphics.white ;;
val draw_num_cc : window_config -> int -> int -> int -> unit = <fun>
```

- Display of a cell containing a mine:

```
# let draw_mine_cc wcf i j =
     open_ccell wcf i j ;
     let cc = wcf.current_box
     in draw_box wcf.current_box ;
        Graphics.set_color Graphics.black ;
        Graphics.fill_circle (cc.x+cc.w/2) (cc.y+cc.h/2) (cc.h/3) ;;
val draw_mine_cc : window_config -> int -> int -> unit = <fun>
```

- Display of a flagged cell containing a mine:

```
# let draw_flag_cc wcf i j =
     close_ccell wcf i j ;
     draw_box wcf.current_box ;
     draw_string_in_box Center "!" wcf.current_box Graphics.blue ;;
val draw_flag_cc : window_config -> int -> int -> unit = <fun>
```

- Display of a wrongly flagged cell:

```
# let draw_cross_cc wcf i j =
    let x,y = wcf.cell (i,j)
    and w,h = wcf.current_box.w, wcf.current_box.h in
    let a=x+w/4 and b=x+3*w/4
```

```
        and c=y+h/4 and d=y+3*h/4
        in Graphics.set_color Graphics.red ;
           Graphics.set_line_width 3 ;
           Graphics.moveto a d ; Graphics.lineto b c ;
           Graphics.moveto a c ; Graphics.lineto b d ;
           Graphics.set_line_width 1 ;;
    val draw_cross_cc : window_config -> int -> int -> unit = <fun>
```

During the game, the choice of the display function to use is done by:

```
# let draw_cell wcf bd i j =
    let cell = bd.(i).(j)
    in match (cell.flag, cell.seen , cell.mined ) with
           (true,_,_)  → draw_flag_cc wcf i j
         | (_,false,_) → draw_closed_cc wcf i j
         | (_,_,true)  → draw_mine_cc wcf i j
         | _           → draw_num_cc wcf i j cell.nbm  ;;
val draw_cell : window_config -> cell array array -> int -> int -> unit =
  <fun>
```

A specialized function displays all the cells at the end of the game. It is slightly different from the previous one as all the cells are taken as opened. Moreover, a red cross indicates the empty cells where the player wrongly put a flag.

```
# let draw_cell_end wcf bd i j =
    let cell = bd.(i).(j)
    in match (cell.flag, cell.mined ) with
           (true,true) → draw_flag_cc wcf i j
         | (true,false) → draw_num_cc wcf i j cell.nbm; draw_cross_cc wcf i j
         | (false,true) → draw_mine_cc wcf i j
         | (false,false) → draw_num_cc wcf i j cell.nbm ;;
val draw_cell_end : window_config -> cell array array -> int -> int -> unit =
  <fun>
```

Display of the other components The state of the flagging mode is indicated by a box that is either at the bottom or on top and that contain either the word `ON` or `OFF`:

```
# let draw_flag_switch wcf on =
    if on  then wcf.flag_box.r <- Bot  else wcf.flag_box.r <- Top ;
    draw_box wcf.flag_box ;
    if on  then draw_string_in_box Center "ON" wcf.flag_box Graphics.red
    else draw_string_in_box Center "OFF" wcf.flag_box Graphics.blue ;;
val draw_flag_switch : window_config -> bool -> unit = <fun>
```

We display the purpose of the flagging button above it:

```
# let draw_flag_title wcf =
    let m = "Flagging" in
    let w,h = Graphics.text_size m in
    let x = (wcf.main_box.w-w)/2
    and y0 = wcf.dialog_box.y+wcf.dialog_box.h in
    let y = y0+(wcf.main_box.h-(y0+h))/2
    in Graphics.moveto x y ;
       Graphics.draw_string m ;;
val draw_flag_title : window_config -> unit = <fun>
```

During the game, the number of empty cells left to be opened and the number of cells to flag are displayed in the dialog box, to the left and right of the flagging mode button.

```
# let print_score wcf nbcto nbfc =
    erase_box wcf.d1_box ;
    draw_string_in_box Center (string_of_int nbcto) wcf.d1_box Graphics.blue ;
    erase_box wcf.d2_box ;
    draw_string_in_box Center (string_of_int (wcf.cf.nbmines-nbfc)) wcf.d2_box
     ( if nbfc>wcf.cf.nbmines then Graphics.red else Graphics.blue ) ;;
val print_score : window_config -> int -> int -> unit = <fun>
```

To draw the initial mine field, we need to draw (number of rows) × (number of columns) times the same closed cell. It is always the same drawing, but it may take a long time, as it is necessary to draw a rectangle as well as four trapezoids. To speed up this initialization, we draw only one cell, take the bitmap corresponding to this drawing, and paste this bitmap into every cell.

```
# let draw_field_initial wcf =
    draw_closed_cc wcf 0 0 ;
    let cc = wcf.current_box  in
    let bitmap = draw_box cc ; Graphics.get_image cc.x cc.y cc.w cc.h  in
    let draw_bitmap (i,j) = let x,y=wcf.cell (i,j)
                            in Graphics.draw_image bitmap x y
    in iter_cells wcf.cf draw_bitmap  ;;
val draw_field_initial : window_config -> unit = <fun>
```

At the end of the game, we open the whole mine field while putting a red cross on cells wrongly flagged:

```
# let draw_field_end wcf bd =
    iter_cells wcf.cf (fun (i,j) → draw_cell_end wcf bd i j)  ;;
val draw_field_end : window_config -> cell array array -> unit = <fun>
```

Finally, the main display function called at the beginning of the game opens the graphical context and displays the initial state of all the components.

```
# let open_wcf wcf =
    Graphics.open_graph ( " " ^ (string_of_int wcf.main_box.w) ^ "x" ^
                          (string_of_int wcf.main_box.h)               ) ;
    draw_box wcf.main_box ;
    draw_box wcf.dialog_box ;
    draw_flag_switch wcf false ;
    draw_box wcf.field_box ;
    draw_field_initial wcf ;
    draw_flag_title wcf ;
    print_score wcf ((wcf.cf.nbrows*wcf.cf.nbcols)-wcf.cf.nbmines) 0 ;;
val open_wcf : window_config -> unit = <fun>
```

Notice that all the display primitives are parameterized by a graphical configuration of type *window_config*. This makes them independent of the layout of the components of our Minesweeper. If we wish to modify the layout, the code still works without any modification, only the configuration needs to be updated.

Interaction with the player

We now list what the player may do:

- he may click on the mode box to change mode (opening or flagging),
- he may click on a cell to open it or flag it,
- he may hit the `'q'` key to quit the game.

Recall that a `Graphic` event (*Graphics.event*) must be associated with a record (*Graphics.status*) that contains the current information on the mouse and keyboard when the event occurs. An interaction with the mouse may happen on the mode button, or on a cell of the mine field. Every other mouse event must be ignored. In order to differentiate these mouse events, we create the type:

```
# type clickon = Out | Cell of (int*int) | SelectBox  ;;
```

Also, pressing the mouse button and releasing it are two different events. For a click to be valid, we require that both events occur on the same component (the flagging mode button or a cell of the mine field).

```
# let locate_click wcf st1 st2 =
    let clickon_of st =
      let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y
      in if x>=wcf.flag_box.x && x<=wcf.flag_box.x+wcf.flag_box.w &&
            y>=wcf.flag_box.y && y<=wcf.flag_box.y+wcf.flag_box.h
         then SelectBox
         else let (x2,y2) = wcf.coor (x,y)
              in if x2>=0 && x2<wcf.cf.nbcols && y2>=0 && y2<wcf.cf.nbrows
                 then Cell (x2,y2) else Out
    in
```

```
    let r1=clickon_of st1 and r2=clickon_of st2
    in if r1=r2 then r1 else Out ;;
val locate_click :
  window_config -> Graphics.status -> Graphics.status -> clickon = <fun>
```

The heart of the program is the event waiting and processing loop defined in the function `loop`. It is similar to the function `skel` described page 133, but specifies the mouse events more precisely. The loop ends when:

- the player presses the `q` or `Q` key, meaning that he wants to end the game;
- the player opens a cell containing a mine, then he loses;
- the player has opened all the cell that are empty, then he wins the game.

We gather in a record of type *minesw_cf* the information useful for the interface:

```
# type minesw_cf =
    { wcf : window_config; bd : cell array array;
      mutable nb_flagged_cells : int;
      mutable nb_hidden_cells : int;
      mutable flag_switch_on : bool } ;;
```

The meaning of the fields is:

- `wcf`: the graphical configuration;
- `bd`: the board;
- `flag_switch_on`: a boolean indicating whether flagging mode or opening mode is on;
- `nb_flagged_cells`: the number of flagged cells;
- `nb_hidden_cells`: the number of empty cells left to open;

The main loop is implemented this way:

```
# let loop d f_init f_key f_mouse f_end =
    f_init ();
    try
      while true do
        let st = Graphics.wait_next_event
                   [Graphics.Button_down;Graphics.Key_pressed]
        in if st.Graphics.keypressed  then f_key st.Graphics.key
           else let st2 = Graphics.wait_next_event [Graphics.Button_up]
                in f_mouse (locate_click d.wcf st st2)
      done
    with End → f_end ();;
val loop :
  minesw_cf ->
  (unit -> 'a) -> (char -> 'b) -> (clickon -> 'b) -> (unit -> unit) -> unit =
  <fun>
```

The initialization function, cleanup function and keyboard event processing function are very simple.

```
# let d_init d () = open_wcf d.wcf
  let d_end  () = Graphics.close_graph()
  let d_key c = if c='q' || c='Q' then raise End;;
val d_init : minesw_cf -> unit -> unit = <fun>
val d_end : unit -> unit = <fun>
val d_key : char -> unit = <fun>
```

However, the mouse event processing function requires the use of some auxiliary functions:

- `flag_cell`: when clicking on a cell with flagging mode on.
- `ending`: when ending the game. The whole mine field is revealed, we display a message indicating whether the game was won or lost, and we wait for a mouse or keyboard event to quit the application.
- `reveal`: when clicking on a cell with opening mode on (*i.e.* flagging mode off).

```
# let flag_cell d i j =
    if d.bd.(i).(j).flag
    then ( d.nb_flagged_cells <- d.nb_flagged_cells -1;
           d.bd.(i).(j).flag <- false )
    else ( d.nb_flagged_cells <- d.nb_flagged_cells +1;
           d.bd.(i).(j).flag <- true );
    draw_cell d.wcf d.bd i j;
    print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;;
val flag_cell : minesw_cf -> int -> int -> unit = <fun>

# let ending d str =
    draw_field_end d.wcf d.bd;
    erase_box  d.wcf.flag_box;
    draw_string_in_box Center str d.wcf.flag_box Graphics.black;
    ignore(Graphics.wait_next_event
             [Graphics.Button_down;Graphics.Key_pressed]);
    raise End;;
val ending : minesw_cf -> string -> 'a = <fun>

# let reveal d i j =
    let reveal_cell (i,j) =
      d.bd.(i).(j).seen <- true;
      draw_cell d.wcf d.bd i j;
      d.nb_hidden_cells <- d.nb_hidden_cells -1
    in
      List.iter reveal_cell (cells_to_see d.bd d.wcf.cf (i,j));
      print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;
      if d.nb_hidden_cells = 0 then ending d "WON";;
```

```
val reveal : minesw_cf -> int -> int -> unit = <fun>
```

The mouse event processing function matches a value of type *clickon*.

```
# let d_mouse d click = match click with
      Cell (i,j) →
        if d.bd.(i).(j).seen then ()
        else if d.flag_switch_on then flag_cell d i j
        else if d.bd.(i).(j).flag then ()
        else if d.bd.(i).(j).mined then ending d "LOST"
        else reveal d i j
    | SelectBox →
        d.flag_switch_on <- not d.flag_switch_on;
        draw_flag_switch d.wcf d.flag_switch_on
    | Out → () ;;
val d_mouse : minesw_cf -> clickon -> unit = <fun>
```

To create a game configuration, three parameters are needed: the number of columns, the number of rows, and the number of mines.

```
# let create_minesw nb_c nb_r nb_m =
    let nbc = max default_config.nbcols nb_c
    and nbr = max default_config.nbrows nb_r in
    let nbm = min (nbc*nbr) (max 1 nb_m) in
    let cf = { nbcols=nbc ; nbrows=nbr ; nbmines=nbm } in
    generate_seed () ;
    let wcf = make_wcf cf in
    {  wcf = wcf ;
       bd = initialize_board wcf.cf;
       nb_flagged_cells = 0;
       nb_hidden_cells = cf.nbrows*cf.nbcols-cf.nbmines;
       flag_switch_on = false } ;;
val create_minesw : int -> int -> int -> minesw_cf = <fun>
```

The launch function creates a configuration according to the numbers of columns, rows, and mines, before calling the main event processing loop.

```
# let go nbc nbr nbm =
    let d = create_minesw nbc nbr nbm in
    loop d (d_init d) d_key (d_mouse d) (d_end);;
val go : int -> int -> int -> unit = <fun>
```

The function call `go 10 10 10` builds and starts a game of the same size as the one depicted in figure 6.5.

Exercises

This program can be built as a standalone executable program. Chapter 7 explains how to do this. Once it is done, it is useful to be able to specify the size of the game on the command line. Chapter 8 describes how to get command line arguments in an Objective Caml program, and applies it to our minesweeper (see page 236).

Another possible extension is to have the machine play to discover the mines. To do this, one needs to be able to find the safe moves and play them first, then compute the probabilities of presence of a mine and open the cell with the smallest probability.