11
Algebraic data types

Data types can be divided into the *atomic* types (with atomic type constructors like `int` and `bool`) and the *composite* types (with parameterized type constructors like ⟨⟩ `*` ⟨⟩ , ⟨⟩ `list`, and ⟨⟩ `option`).

What is common to all of the built-in composite types introduced so far[1] is that they allow building data structures through the combination of just two methods.

1. *Conjunction*: Multiple components can be conjoined to form a composite value containing *all* of the components.

   For instance, values of pair type, `int * float` say, are formed as the conjunction of two components, the first component an `int` and the second a `float`.

2. *Alternation*: Multiple components can be disjoined, serving as alternatives to form a composite value containing *one* of the values.

   For instance, values of type `int list` are formed as the alternation of two components. One alternative is `[]`; the other is the "cons" (itself a conjunction of a component of type `int` and a component of type `int list`).

Data types built by conjunction and disjunction are called ALGEBRAIC DATA TYPES.[2] As mentioned, we've seen several examples already, as built-in composite data types. But why should the power of algebraic data types be restricted to built-in types? Such a simple and elegant construction like algebraic types could well be a foundational construct of the language, not only to empower programmers using the language but also to provide a foundation for the built-in constructs themselves.

OCaml inherits from its antecedents (especially, the Hope programming language developed at the University of Edinburgh, the university that brought us ML as well) the ability to define new algebraic data types as user code.

[1] The exception is the composite type of functions. Functions are the rare case of a composite type in OCaml not structured as an algebraic data type as defined below.

[2] Algebra is the mathematical study of structures that obey certain laws. Typical of algebras is to form such structures by operations that have exactly the duality of conjunction and alternation found here. For instance, arithmetic algebras have multiplication and addition as, respectively, the conjunction and alternation operators. Boolean algebras have logical conjunction ('and') and disjunction ('or'). Set algebras have cross-product and union. The term *algebraic data type* derives from this connection to these structured algebras.

Let's start with a simple example based on genome processing, exemplifying the use of alternation. DNA sequences are long sequences composed of only four base amino acids: guanine (G), cytosine (C), adenine (A), and thymine (T).

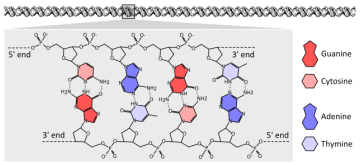

Figure 11.1: DNA carries information encoded as sequences of four amino acids.

We can define an algebraic data type for the DNA bases via alternation. The type, called `base`, will have four value constructors corresponding to the four base letters. The alternatives are separated by vertical bars (`|`). Here is the definition of the `base` type, introduced by the keyword `type`:

```
# type base = G | C | A | T ;;
type base = G | C | A | T
```

This kind of type declaration defines a VARIANT TYPE, which lists a set of alternatives, variant ways of building elements of the type: `A` *or* `T` *or* `C` *or* `G`.[3] Having defined the `base` type, we can refer to values of that type.

```
# A ;;
- : base = A
# G ;;
- : base = G
```

[3] Using argumentless variants in this way serves the purpose of enumerated types in other languages – `enum` in C, C derivatives, Java, and Perl, for instance. Variants thus generalize enumerated types.

As with all composite types, computations that depend on the particular values of the type use pattern-matching to structure the cases. For instance, each DNA base has a complementary base: A and T are complementary, as are G and C. A function to return the complement of a base uses pattern-matching to individuate the cases:

```
# let comp_base bse =
#   match bse with
#   | A -> T
#   | T -> A
#   | G -> C
#   | C -> G ;;
val comp_base : base -> base = <fun>
# comp_base G ;;
- : base = C
```

Variants correspond to the alternation approach to building composite values. The conjunction approach is enabled by allowing the alternative value constructors to take an argument of a specified type. That argument itself can conjoin components by tupling.

As an example, DNA sequences themselves can be implemented as an algebraic data type that we'll call `dna`. Taking inspiration from the `list` type for sequences, DNA sequences can be categorized into two alternatives, two variants – the empty sequence, for which we will use the value constructor `Nil`; and non-empty sequences, for which we will use the value constructor `Cons`. The `Cons` constructor will take two

arguments (uncurried), one for the first `base` in the sequence and one for the rest of the `dna` sequence.[4]

```
# type dna =
#   | Nil
#   | Cons of base * dna ;;
type dna = Nil | Cons of base * dna
```

The `Cons` constructor takes two arguments (using tuple notation), the first of type `base` and the second of type `dna`. It thus serves to conjoin a `base` element and another `dna` sequence.

Having defined this new type, we can construct values of that type:

```
# let seq = Cons (A, Cons (G, Cons (T, Cons (C, Nil)))) ;;
val seq : dna = Cons (A, Cons (G, Cons (T, Cons (C, Nil))))
```

and pattern-match against them:

```
# let first_base =
#   match seq with
#   | Cons (x, _) -> x
#   | Nil -> failwith "empty sequence" ;;
val first_base : base = A
```

The `dna` type is defined recursively,[5] as one of its variants (`Cons`) includes another value of the same type. By using recursion, we can define data types whose values can be of arbitrary size.

To process data values of arbitrary size, recursive functions are an ideal match. A function to construct the complement of an entire DNA sequence is naturally recursive.

```
# let rec complement seq =
#   match seq with
#   | Nil -> Nil
#   | Cons (b, seq) -> Cons (comp_base b, complement seq) ;;
val complement : dna -> dna = <fun>

# complement seq ;;
- : dna = Cons (T, Cons (C, Cons (A, Cons (G, Nil))))
```

11.1 Built-in composite types as algebraic types

The `dna` type looks for all the world just like the `list` type built into OCaml, except for the fact that its elements are always of type `base`. Indeed, our choice of names of the value constructors (`Nil` and `Cons`) emphasizes the connection.

In fact, many of the built-in composite types can be implemented as algebraic data types in this way. Boolean values are essentially a kind of enumerated type, hence algebraic.[6]

```
# type bool_ = True | False ;;
type bool_ = True | False
```

[4] 🕮 There is a subtle distinction concerning when type constructors take a single tuple argument or multiple arguments written with tuple notation. For the most part, the issue can be ignored, so long as the type definition doesn't place the argument sequence within parentheses. For the curious, see the "Note on tupled constructors" in the OCaml documentation.

[5] In *value* definitions (with `let`), recursion must be marked explicitly with the `rec` keyword. In *type* definitions, no such explicit marking is required, and in fact nonrecursive definitions can only be formed using distinct type names. This design decision was presumably motivated by the ubiquity of recursive type definitions as compared to recursive value definitions. It's a contentious matter as to whether this quirk of OCaml is a feature or a bug.

[6] We name the type `bool_` so as not to shadow the built-in type `bool`. Similarly for the underscore versions `list_` and `option_` below.

Value constructors in defined algebraic types are restricted to starting with capital letters in OCaml. The built-in type differs only in using lower case constructors `true` and `false`.

We've already seen an algebraic type implementation of `base` lists. Similar implementations could be generated for lists of other types.

```
# type int_list = INil | ICons of int * int_list ;;
type int_list = INil | ICons of int * int_list
# type float_list = FNil | FCons of float * float_list ;;
type float_list = FNil | FCons of float * float_list
```

Following the edict of irredundancy, we'd prefer not to write this same code repeatedly, differing only in the type of the list elements. Fortunately, variant type declarations can be polymorphic.

```
# type 'a list_ = Nil | Cons of 'a * 'a list_ ;;
type 'a list_ = Nil | Cons of 'a * 'a list_
```

In polymorphic variant data type declarations like this, a new type constructor (`list_` in this case) is defined that takes a type argument (here, the type variable `'a`). The type constructor is always postfix, like the built-in constructors `list` and `option` that you've already seen.[7]

Option types can be viewed as a polymorphic variant type with two constructors for the `None` and `Some` cases.

```
# type 'a option_ = None | Some of 'a ;;
type 'a option_ = None | Some of 'a
```

The point of seeing these alternative implementations of the built-in composite types (booleans, lists, options) is not that one would actually *use* these implementations. That would flout the edict of irredundancy. And the reimplementations of lists and options don't benefit from the concrete syntax niceties of the built-in versions; no infix `::` for instance, or bracketed lists. Rather than defining a `dna` type in this way, in a real application we'd just use the `base list` type. If a name for this type is desired the type name `dna` can be defined by

```
type dna = base list ;;
```

The point instead is to demonstrate the power of algebraic data type definitions and show that even more of the language can be viewed as syntactic sugar for pre-provided user code. Thus, the language can again be seen as deploying a small core of basic notions to build up a highly expressive medium.

11.2 Example: Boolean document search

The variant type definitions in this chapter aren't the first examples of algebraic type definitions you've seen. In Section 7.4, we noted that record types were user-defined types, defined with the `type` keyword, as well.

[7] If we need a type constructor that takes more than one type as an argument, we use the cross-product type notation, as in the `('key, 'value) dictionary` type defined in Section 11.3.

Record types are a kind of dual to variant types. Instead of starting with alternation – this *or* that *or* the other – record types start with conjunction – this *and* that *and* the other.

As an example, consider a data type for documents. A document will be made up of a list of words (each a string), as well as some metadata about the document, perhaps its title, author, and so forth. For this example, we'll stick just to titles, so an appropriate type definition would be

```
# type document = { title : string;
#                   words : string list } ;;
type document = { title : string; words : string list; }
```

A corpus of such documents can be implemented as a `document list`. We build a small corpus of first lines of novels.[8]

```
# let first_lines : document list =  (* output suppressed *)
#   [ {title = "Moby Dick";
#      words = tokenize
#                "Call me Ishmael ."};
#     {title = "Pride and Prejudice";
#      words = tokenize
#                "It is a truth universally acknowledged , \
#                 that a single man in possession of a good \
#                 fortune must be in want of a wife ."};
#     {title = "1984";
#      words = tokenize
#                "It was a bright cold day in April , and \
#                 the clocks were striking thirteen ."};
#     {title = "Great Gatsby";
#      words = tokenize
#                "In my younger and more vulnerable years \
#                 my father gave me some advice that I've \
#                 been turning over in my mind ever since ."}
#   ] ;;
```

We might want to query for documents with particular patterns of words. A boolean query allows for different query types: requesting documents in which a particular word occurs; or (inductively) documents that satisfy both one query and another query; or documents that satisfy either one query or another query. We instantiate the idea in a variant type definition.

```
# type query =
#   | Word of string
#   | And of query * query
#   | Or of query * query ;;
type query = Word of string | And of query * query | Or of query *
  query
```

To evaluate such queries against a document, we'll write a function `eval : document -> query -> bool`, which should return `true` just in case the document satisfies the query.

[8] As an aid in building a document corpus, it will be useful to have a function `tokenize : string -> string list` that splits up a string into its component words (here defined as any characters separated by whitespace). We use some functions from the `Str` library module, made available using the `#load` directive to the REPL, to split up the string.

```
# #load "str.cma" ;;
# let tokenize : string -> string list =
#   Str.split (Str.regexp "[ \t\n]+") ;;
val tokenize : string -> string list = <fun>
```

Did you notice the use of partial application?

We've also suppressed the output for this REPL input to save space, as indicated by the `(* output suppressed *)` comment here and elsewhere.

```
let rec eval ({title; words} : document)
             (q : query)
           : bool = ...
```

Note the use of pattern-matching right in the header line, as well as the use of field punning to simplify the pattern.

The evaluation of the query depends on its structure, so we'll want to match on that.

```
let rec eval ({title; words} : document)
             (q : query)
           : bool =
  match q with
  | Word word -> ...
  | And (q1, q2) -> ...
  | Or (q1, q2) -> ...
```

For the first variant, we merely check that the `word` occurs in the list of `words`:

```
let rec eval ({title; words} : document)
             (q : query)
           : bool =
  match q with
  | Word word -> List.mem word words
  | And (q1, q2) -> ...
  | Or (q1, q2) -> ...
```

(The function `List.mem : 'a -> 'a list -> bool` is useful here, a good reason to familiarize yourself with the rest of the `List` library module.)

What about the other variants? In these cases, we'll want to recursively evaluate the subparts of the query (`q1` and `q2`) against the same document. We've already decomposed the document into its components `title` and `words`. We could reconstruct the document as needed for the recursive evaluations:

```
let rec eval ({title; words} : document)
             (q : query)
           : bool =
  match q with
  | Word word -> List.mem word words
  | And (q1, q2) -> (eval {title; words} q1)
                    && (eval {title; words} q2)
  | Or (q1, q2) -> (eval {title; words} q1)
                   || (eval {title; words} q2) ;;
```

but this seems awfully verbose. We refer to `{title; words}` four different times. It would be helpful if we could both pattern match against the document argument and name it as a whole as well. OCaml provides a special pattern constructed as

⟨*pattern*⟩ `as` ⟨*var*⟩

for just such cases. Such a pattern both pattern matches against the ⟨*pattern*⟩ as well as binding the ⟨*var*⟩ to the expression being matched against as a whole. We use this technique both to provide a name for the document as a whole (`doc`) and to extract its components. (Once we have a variable `doc` for the document as a whole, we no longer need to refer to `title`, so we use an anonymous variable instead.)

```
let rec eval ({words; _} as doc : document)
             (q : query)
           : bool =
    match q with
    | Word word -> List.mem word words
    | And (q1, q2) -> (eval doc q1)  && (eval doc q2)
    | Or (q1, q2) -> (eval doc q1) || (eval doc q2) ;;
```

That's better. But we're still calling `eval doc` four times on different subqueries. We can abstract that function and reuse it; call it `eval'`:

```
let eval ({words; _} as doc : document)
         (q : query)
       : bool =
  let rec eval' (q : query) : bool =
    match q with
    | Word word -> List.mem word words
    | And (q1, q2) -> (eval' q1) && (eval' q2)
    | Or (q1, q2) -> (eval' q1) || (eval' q2) in
  ... ;;
```

There's an important idea hidden here, which follows from the scoping rules of OCaml. Because the `eval'` definition falls within the scope of the definition of `eval` and the associated variables `words` and `q`, those variables are available in the body of the `eval'` definition. And in fact, we make use of that fact by referring to `words` in the first pattern-match. (The outer `q` is actually shadowed by the inner `q`, so it isn't referred to in the body of the `eval'` definition. The occurrence of `q` in the `match q` is a reference to the `q` argument of `eval'`.)

Now that we have `eval'` defined it suffices to call it on the main query and let the recursion do the rest. At this point, however, the alternative variable name `doc` is no longer referenced, and can be eliminated.

```
# let eval ({words; _} : document)
#          (q : query)
#        : bool =
#   let rec eval' (q : query) : bool =
#     match q with
#     | Word word -> List.mem word words
#     | And (q1, q2) -> (eval' q1) && (eval' q2)
#     | Or (q1, q2) -> (eval' q1) || (eval' q2) in
#   eval' q ;;
val eval : document -> query -> bool = <fun>
```

Let's try it on some sample queries. We'll use the first line of *The Great Gatsby*.

```
# let gg = nth first_lines 3 ;;  (* output suppressed *)

# eval gg (Word "the") ;;
- : bool = false
# eval gg (Word "and") ;;
- : bool = true
# eval gg (And ((Word "the"), (Word "and"))) ;;
- : bool = false
# eval gg (Or ((Word "the"), (Word "and"))) ;;
- : bool = true
```

Now, we return to the original goal, to search among a whole corpus of documents for those satisfying a query. The function `eval_all : document list -> query -> string list` will return the titles of all documents in the `document list` that satisfy the `query`.

The `eval_all` function should be straightforward to write, as it involves filtering the document list for those satisfying the query, then extracting their titles. The `filter` and `map` list-processing functions are ideal for this.

```
# let eval_all (docs : document list)
#              (q : query)
#            : string list =
#   List.map (fun doc -> doc.title)
#            (List.filter (fun doc -> (eval doc q))
#                         docs) ;;
val eval_all : document list -> query -> string list = <fun>
```

We start with the `docs`, filter them with a function that applies `eval` to select only those that satisfy the query, and then map a function over them to extract their titles.

From a readability perspective, it is unfortunate that the description of what the code is doing – start with the corpus, then filter, then map – is "inside out" with respect to how the code reads. This follows from the fact that in OCaml, functions come before their arguments in applications, whereas in this case, we like to think about a data object followed by a set of functions that are applied to it. A language with backwards application would be able to structure the code in the more readable manner.

Happily, the `Stdlib` module provides a BACKWARDS APPLICATION infix operator `|>` for just such occasions.

```
# succ 3 ;;
- : int = 4
# 3 |> succ ;;       (* start with 3; increment *)
- : int = 4
# 3 |> succ          (* start with 3; increment; ... *)
```

```
#   |> (( * ) 2) ;;  (* ... and double *)
- : int = 8
```

Exercise 87

What do you expect the type of |> is?

Exercise 88

How could you define the backwards application operator |> as user code?

Taking advantage of the backwards application operator can make the code considerably more readable. Instead of

```
List.filter (fun doc -> (eval doc q))
            docs
```

we can start with `docs` and then filter it:

```
docs
|> List.filter (fun doc -> (eval doc q))
```

Then we can map the title extraction function over the result:

```
docs
|> List.filter (fun doc -> (eval doc q))
|> List.map (fun doc -> doc.title)
```

The final definition of `eval_all` is then

```
# let eval_all (docs : document list)
#              (q : query)
#            : string list =
#   docs
#   |> List.filter (fun doc -> (eval doc q))
#   |> List.map (fun doc -> doc.title) ;;
val eval_all : document list -> query -> string list = <fun>
```

Some examples:

```
# eval_all first_lines (Word "and") ;;
- : string list = ["1984"; "Great Gatsby"]
# eval_all first_lines (Word "me") ;;
- : string list = ["Moby Dick"; "Great Gatsby"]
# eval_all first_lines (And (Word "and", Word "me")) ;;
- : string list = ["Great Gatsby"]
# eval_all first_lines (Or (Word "and", Word "me")) ;;
- : string list = ["Moby Dick"; "1984"; "Great Gatsby"]
```

The change in readability from using backwards application has a moral. Concrete syntax can make a big difference in the human usability of a programming language. The addition of a backwards application adds not a jot to the expressive power of the language, but when used appropriately it can dramatically reduce the cognitive load on a human reader.[9]

[9] Not coincidentally, natural languages often allow alternative orders for phrases for just this same goal of moving "heavier" phrases to the right. For example, the normal order for verb phrases with the verb "give" places the object before the recipient, as in "Arden gave the book to Bellamy". But when the object is very "heavy" (long and complicated), it sounds better to place the object later, as in "Arden gave to Bellamy every last book in the P. G. Wodehouse collection." Backwards application gives us this same flexibility, to move "heavy" expressions (like complicated functions) later in the code.

11.3 Example: Dictionaries

A dictionary is a data structure that manifests a relationship between a set of *keys* and their associated *values*. In an English dictionary, for instance, the keys are the words of the language and the associated values are their definitions. But dictionaries can be used in a huge variety of applications.

A dictionary data type will depend on the types of the keys and the values. We'll want to define the type, then, as polymorphic – a `('key, 'value) dictionary`.[10] One approach (an exceptionally poor one as it will turn out, but bear with us) is to store the keys and values as separate equal-length lists in two record fields.

[10] Names of type variables are arbitrary, so we might as well use that ability to give good mnemonic names to them – `'key` and `'value` instead of `'a` and `'b` – following the edict of intention in making our intentions clear to readers of the code.

```
# type ('key, 'value) dictionary = { keys : 'key list;
#                                    values : 'value list } ;;
type ('key, 'value) dictionary = { keys : 'key list; values :
  'value list; }
```

Looking up an entry in the dictionary by key, returning the corresponding value, can be performed in a few ways. Here's one:

```
# let rec lookup ({keys; values} : ('key, 'value) dictionary)
#                (request : 'key)
#              : 'value option =
#   match keys, values with
#   | [], [] -> None
#   | key :: keys, value :: values ->
#       if key = request then Some value
#       else lookup {keys; values} request ;;
Lines 4-8, characters 0-34:
4 | match keys, values with
5 | | [], [] -> None
6 | | key :: keys, value :: values ->
7 | if key = request then Some value
8 | else lookup {keys; values} request...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
([], _::_)
val lookup : ('key, 'value) dictionary -> 'key -> 'value option =
  <fun>
```

The problem with this dictionary representation is obvious. The entire notion of a dictionary assumes that for each key there is a single value. But this approach to implementing dictionaries provides no such guarantee. An illegal dictionary – like `{keys = [1; 2; 3]; values = ["first"; "second"]}`, in which one of the keys has no value – is representable. In such cases, the `lookup` function will raise an exception.

```
# let bad_dict = {keys = [1; 2; 3];
#                 values = ["first"; "second"]} ;;
```

```
val bad_dict : (int, string) dictionary =
  {keys = [1; 2; 3]; values = ["first"; "second"]}
# lookup bad_dict 4 ;;
Exception: Match_failure ("//toplevel//", 4, 0).
# lookup bad_dict 3 ;;
Exception: Match_failure ("//toplevel//", 4, 0).
```

Adding additional match cases merely postpones the problem.

```
# let rec lookup ({keys; values} : ('key, 'value) dictionary)
#                (request : 'key)
#              : 'value option =
#   match keys, values with
#   | [], _
#   | _, [] -> None
#   | key :: keys, value :: values ->
#       if key = request then Some value
#       else lookup {keys; values} request ;;
val lookup : ('key, 'value) dictionary -> 'key -> 'value option =
  <fun>
# lookup bad_dict 4 ;;
- : string option = None
# lookup bad_dict 3 ;;
- : string option = None
```

The function still allows data structures that do not express legal dictionaries to be used. Indeed, we can no longer even distinguish between simple cases of lookup of a missing key and problematic cases of lookup in an ill-formed dictionary structure.

A better dictionary design would make such illegal structures impossible to even represent. This idea is important enough for its own edict.

Edict of prevention:
Make the illegal inexpressible.

We've seen this idea before in the small. It's the basis of type checking itself, which allows the use of certain values only with functions that are appropriate to apply to them – integers with integer functions, booleans with boolean functions – preventing all other uses. In a strongly typed language like OCaml, illegal operations, like applying an integer function to a boolean value, simply can't be expressed as valid well-typed code.

The edict of prevention[11] challenges us to find an alternative structure in which this kind of mismatch between the keys and values can't occur. Such a structure may already have occurred to you. Instead of thinking of a dictionary as a *pair of lists* of keys and values, we can think of it as a *list of pairs* of keys and values.[12]

```
# type ('key, 'value) dict_entry =
#   { key : 'key; value : 'value }
```

[11] This idea has a long history in functional programming with algebraic data types, but seen in its crispest form is likely due to Yaron Minsky, who phrases it as "Make illegal states unrepresentable." Ben Feldman uses "Make impossible states impossible." But the idea dates back to at least the beginnings of statically typed programming languages. By referring to inexpressibility, rather than unrepresentability, we generalize the notion to include cases we consider in Chapter 12.

[12] An idiosyncrasy of OCaml requires that the dictionary type be defined in stages in this way, rather than all at once as

```
# type ('key, 'value) dictionary =
#   { key : 'key; value : 'value } list ;;
Line 2, characters 31-35:
2 | { key : 'key; value : 'value } list ;;
                                   ^^^^
Error: Syntax error
```

The use of `and` to combine multiple type definitions into a single simultaneous definition isn't required here, but is when the type definitions are mutually recursive.

```
# and ('key, 'value) dictionary =
#   ('key, 'value) dict_entry list ;;
type ('key, 'value) dict_entry = { key : 'key; value : 'value; }
and ('key, 'value) dictionary = ('key, 'value) dict_entry list
```

The type system will now guarantee that every dictionary is a list whose elements each have a key and a value. A dictionary with unequal numbers of keys and values is *not even expressible*. The `lookup` function can still recur through the pairs, looking for the match:

```
# let rec lookup (dict : ('key, 'value) dictionary)
#                (request : 'key)
#              : 'value option =
#   match dict with
#   | [] -> None
#   | {key; value} :: tl ->
#       if key = request then Some value
#       else lookup tl request ;;
val lookup : ('key, 'value) dictionary -> 'key -> 'value option =
  <fun>

# let good_dict = [{key = 1; value = "one"};
#                  {key = 2; value = "two"};
#                  {key = 3; value = "three"}] ;;
val good_dict : (int, string) dict_entry list =
  [{key = 1; value = "one"}; {key = 2; value = "two"};
   {key = 3; value = "three"}]
# lookup good_dict 3 ;;
- : string option = Some "three"
# lookup good_dict 4 ;;
- : string option = None
```

In this particular case, changing the structure of dictionaries to make the illegal inexpressible also very slightly simplifies the lookup code as well. But even if pursuing the edict of prevention makes code a bit more complex, it can be well worth the trouble in preventing bugs from arising in the first place.

Not all illegal states can be prevented by making them inexpressible through the structuring of the types. For instance, this updated dictionary structure still allows dictionaries that are ill-formed in allowing the same key to occur more than once. We'll return to this issue when we further apply the edict of prevention in Chapter 12.

Problem 89

The game of mini-poker is played with just six playing cards: You use only the face cards (king, queen, jack) of the two suits spades and diamonds. There is a ranking on the cards: Any spade is better than any diamond, and within a suit, the cards from best to worst are king, queen, jack.

In this two-player game, each player picks a single card at random, and the player with the better card wins.

For the record, it's a terrible game.

Provide appropriate type definitions to represent the cards used in the game. It should contain structured information about the suit and value of the cards.

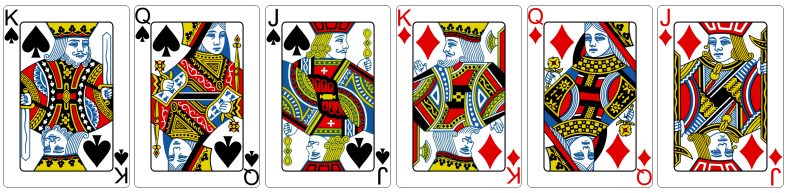

Figure 11.2: The cards of mini-poker, depicted in order from best to worst.

Problem 90

What is an appropriate type for a function `better` that determines which of two cards is "better" in the context of mini-poker, returning `true` if and only if the first card is better than the second?

Problem 91

Provide a definition of the function `better`.

11.4 Example: Arithmetic expressions

One of the elegancies admitted by the generality of algebraic data types is their use in capturing languages.

By way of example, a language of simple integer arithmetic expressions can be defined by the following grammar, written in Backus-Naur form as described in Section 3.1.

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle integer \rangle \\
& | & \langle expr_1 \rangle \texttt{ + } \langle expr_2 \rangle \\
& | & \langle expr_1 \rangle \texttt{ - } \langle expr_2 \rangle \\
& | & \langle expr_1 \rangle \texttt{ * } \langle expr_2 \rangle \\
& | & \langle expr_1 \rangle \texttt{ / } \langle expr_2 \rangle \\
& | & \texttt{~-} \langle expr \rangle
\end{array}
$$

(We'll take this to define the abstract syntax of the language. Concrete syntax notions like precedence and associativity of the operators and parentheses for disambiguating structure will be left implicit in the usual way.)

We can define a type for abstract syntax trees for these arithmetic expressions as an algebraic data type. The definition follows the grammar almost trivially, one variant for each line of the grammar.

```
# type expr =
#   | Int of int
#   | Plus of expr * expr
#   | Minus of expr * expr
#   | Times of expr * expr
#   | Div of expr * expr
#   | Neg of expr ;;
type expr =
    Int of int
  | Plus of expr * expr
  | Minus of expr * expr
```

```
  | Times of expr * expr
  | Div of expr * expr
  | Neg of expr
```

The arithmetic expression given in OCaml concrete syntax as `(3 + 4) * ~- 5` corresponds to the following value of type `expr`:

```
# Times (Plus (Int 3, Int 4), Neg (Int 5)) ;;
- : expr = Times (Plus (Int 3, Int 4), Neg (Int 5))
```

A natural thing to do with expressions is to evaluate them. The recursive definition of the `expr` type lends itself to recursive evaluation of values of that type, as in this definition of a function `eval : expr -> int`.

```
# let rec eval (exp : expr) : int =
#   match exp with
#   | Int v        -> v
#   | Plus (x, y)  -> (eval x) + (eval y)
#   | Minus (x, y) -> (eval x) - (eval y)
#   | Times (x, y) -> (eval x) * (eval y)
#   | Neg x        -> ~- (eval x) ;;
Lines 2-7, characters 0-29:
2 | match exp with
3 | | Int v        -> v
4 | | Plus (x, y)  -> (eval x) + (eval y)
5 | | Minus (x, y) -> (eval x) - (eval y)
6 | | Times (x, y) -> (eval x) * (eval y)
7 | | Neg x        -> ~- (eval x)...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Div (_, _)
val eval : expr -> int = <fun>
```

Helpfully, the interpreter warns us of a missing case in the match. One of the variants in the algebraic type definition, division, is not covered by the match. A key feature of defining variant types is that the interpreter can perform these kinds of checks on your behalf. The oversight is easily corrected.

```
# let rec eval (exp : expr) : int =
#   match exp with
#   | Int v        -> v
#   | Plus (x, y)  -> eval x + eval y
#   | Minus (x, y) -> eval x - eval y
#   | Times (x, y) -> eval x * eval y
#   | Div (x, y)   -> eval x / eval y
#   | Neg x        -> ~- (eval x) ;;
val eval : expr -> int = <fun>
```

We can test the evaluator with examples like the one above.

```
# eval (Times (Plus (Int 3, Int 4), Neg (Int 5))) ;;
- : int = -35
```

```
# eval (Int 42) ;;
- : int = 42
# eval (Div (Int 5, Int 0)) ;;
Exception: Division_by_zero.
```

Of course, we already have a way of doing these arithmetic calculations in OCaml. We can just type the expressions into OCaml directly using OCaml's concrete syntax.

```
# (3 + 4) * ~- 5 ;;
- : int = -35
# 42 ;;
- : int = 42
# 5 / 0 ;;
Exception: Division_by_zero.
```

So what use is this kind of thing?

This evaluator is not trivial. By making the evaluation of this language explicit, we have the power to change the language to diverge from the language it is implemented in. For instance, OCaml's integer division truncates the result towards zero. But maybe we'd rather round to the nearest integer? We can implement the evaluator to do that instead.

Exercise 92

Define a version of `eval` that implements a different semantics for the expression language, for instance, by rounding rather than truncating integer divisions.

Exercise 93

Define a function `e2s : expr -> string` that returns a string that represents the fully parenthesized concrete syntax for the argument expression. For instance,

```
# e2s (Times (Plus (Int 3, Int 4), Neg (Int 5))) ;;
- : string = "((3 + 4) * (~- 5))"
# e2s (Int 42) ;;
- : string = "42"
# e2s (Div (Int 5, Int 0)) ;;
- : string = "(5 / 0)"
```

The opposite process, recovering abstract syntax from concrete syntax, is called parsing. More on this in the final project (Chapter A).

11.5 Problem section: Binary trees

Trees are a class of data structures that store values of a certain type in a hierarchically structured manner. They constitute a fundamental data structure, second only perhaps to lists in their repurposing flexibility. Indeed, the arithmetic expressions of Section 11.4 are a kind of tree structure.

In this section, we concentrate on a certain kind of polymorphic BINARY TREE, a kind of tree whose nodes have distinct left and right subtrees, possibly empty. Some examples can be seen in Figure 11.3. A binary tree can be an empty tree (depicted with a bullet symbol (•)

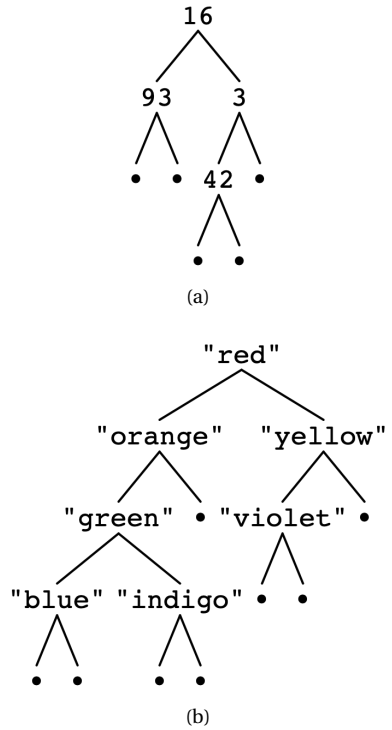

Figure 11.3: Two trees: (a) an integer tree, and (b) a string tree.

in the diagrams), or a node that stores a single value (of type `'a`, say) along with two subtrees, referred to as the left and right subtrees.

A polymorphic binary tree type can thus be defined by the following algebraic data type definition:

```
# type 'a bintree =
#   | Empty
#   | Node of 'a * 'a bintree * 'a bintree ;;
type 'a bintree = Empty | Node of 'a * 'a bintree * 'a bintree
```

For instance, the tree of Figure 11.3(a) can be encoded as an instance of an `int bintree` as

```
# let int_bintree =
#   Node (16,
#     Node (93, Empty, Empty),
#     Node (3,
#       Node (42, Empty, Empty),
#       Empty)) ;;
val int_bintree : int bintree =
  Node (16, Node (93, Empty, Empty),
   Node (3, Node (42, Empty, Empty), Empty))
```

Exercise 94

Construct a value `str_bintree` of type `string bintree` that encodes the tree of Figure 11.3(b).

Now let's write a function to sum up all of the elements stored in an integer tree. The natural approach to carrying out the function is to follow the recursive structure of its tree argument.

```
# let rec sum_bintree (t : int bintree) : int =
#   match t with
#   | Empty -> 0
#   | Node (n, left, right) -> n + sum_bintree left
#                                + sum_bintree right ;;
val sum_bintree : int bintree -> int = <fun>
```

Exercise 95

Define a function `preorder` of type `'a bintree -> 'a list` that returns a list of all of the values stored in a tree in PREORDER, that is, placing values stored at a node before the values in the left subtree, in turn before the values in the right subtree. For instance,

```
# preorder int_bintree ;;
- : int list = [16; 93; 3; 42]
```

You'll notice a certain commonality between the `sum_bintree` and `preorder` functions. Both operate by "walking" the tree, traversing it from its root down, recursively operating on the subtrees, and then combining the value stored at a node and the recursively computed values for the subtrees into the value for the tree as a whole. What differs among them is what value to return for empty trees and what function to apply to compute the overall value from the subparts. We

can abstract this tree walk functionality with a function that takes three arguments: (i) the value to use for empty trees, (ii) the function to apply at nodes to the value stored at the node and the values associated with the two subtrees, along with (iii) a tree to walk; it carries out the recursive process on that tree. Since this is a kind of "fold" operation over binary trees, we'll name the function `foldbt`.

Exercise 96

What is the appropriate type for the function `foldbt` just described?

Exercise 97

Define the function `foldbt` just described.

Exercise 98

Redefine the function `sum_bintree` using `foldbt`.

Exercise 99

Redefine the function `preorder` using `foldbt`.

Exercise 100

Define a function `find : 'a bintree -> 'a -> bool` in terms of `foldbt`, such that `find t v` is `true` just in case the value `v` is found somewhere in the tree `t`.

```
# find int_bintree 3 ;;
- : bool = true
# find int_bintree 7 ;;
- : bool = false
```

11.6 Supplementary material

- Lab 5: Variants, algebraic types, and pattern matching
- Problem set A.3: Bignums and RSA encryption
- Lab 6: Recursive algebraic types
- Problem set A.4: Symbolic differentiation