4
Values and types

OCaml is a

- value-based,
- strongly, statically, implicitly typed,
- functional

programming language. In this chapter, we introduce these aspects of the language.

4.1 OCaml expressions have values

The OCaml language is, at its heart, a language for calculating *values*. The expressions of the language specify these values, and the process of calculating the value of an expression is termed EVALUATION. We've already seen examples of OCaml evaluating some simple expressions in Chapter 2:

```
# 3 + 4 * 5 ;;
- : int = 23
# (3 + 4) * 5 ;;
- : int = 35
```

The results of these evaluations are integers, and the output printed by the REPL indicates this by the `int`, about which we'll have more to say shortly.

4.1.1 Integer values and expressions

Integer values are built using a variety of operators and functions. We've seen the standard arithmetic operators for integer addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). Integer negation is with the `~-` operator (a tilde followed by a hyphen), which is kept distinct from the subtraction operator for clarity.

A full set of built-in operators is provided in OCaml's `Stdlib` module, one of a large set of OCaml library modules that provide a range of functions. The `Stdlib` module is OCaml's "standard library" in the sense that the values it provides can be referred to anywhere without any additional qualification, whereas values from other modules require a prefix, for example, `List.length` or `Hashtbl.create`. [1] You'll want to look over the `Stdlib` module documentation to get a sense of what is available.

[1] There is nothing special going on with `Stdlib`. It's just that by default, the `Stdlib` module is "opened", whereas other library modules like `List` and `Hashtbl` are not. The behavior of modules will become clear when they are fully introduced in Chapter 12.

Here are some examples of integer expressions using these operators:

```
# 1001 / 365 ;;               (* # of years in 1001 nights *)
- : int = 2
# 1001 mod 365 ;;             (* # of nights left over *)
- : int = 271
# 1001 - (1001 / 365) * 365 ;; (* ...or alternatively *)
- : int = 271
```

Notice the use of comments to document the intentions behind the calculations.

4.1.2 Floating point values and expressions

In addition to integers, OCaml provides other kinds of values. Real numbers can be represented using a floating point approximation. Floating point literals can be expressed in several ways, using decimal notation (`3.14`), with an exponent (`314e-2`), and even in hexadecimal (`0x1.91eb851eb851fp+1`).

```
# 3.14 ;;
- : float = 3.14
# 314e-2 ;;
- : float = 3.14
# 0x1.91eb851eb851fp+1 ;;
- : float = 3.14
```

Floating point expressions can be built up with a variety of operators, including addition (`+.`), subtraction (`-.`), multiplication (`*.`), division (`/.`), and negation (`~-.`). Again, the `Stdlib` module provides a fuller set, including operators for square root (`sqrt`) and various kinds of rounding (`floor` and `ceil`).

```
# 3.14 *. 2. *. 2. ;; (* area of circle of radius 2 *)
- : float = 12.56
# ~-. 5e10 /. 2.718 ;;
- : float = -18395879323.0316429
```

Notice that the floating point operators are distinct from those for integers. Though this will take some getting used to, the reason for this design decision in the language will become apparent shortly.

Exercise 8
Use the OCaml REPL to calculate the value of the GOLDEN RATIO, a proportion thought to be especially pleasing to the eye (Figure 4.1).

$$\frac{1+\sqrt{5}}{2} \quad .$$

You'll want to use the built in `sqrt` function for floating point numbers. Be careful to use floating point literals and operators. If you find yourself confronted with errors in solving this exercise, come back to it after reading Section 4.2.

Figure 4.1: A rectangle with width and height in the golden ratio.

4.1.3 *Character and string values*

As in many programming languages, text is represented as strings of CHARACTERS. Character literals are given in single quotes, for instance, `'a'`, `'X'`, `'3'`. Certain special characters can be specified only by escaping them with a backslash, for instance, the single-quote character itself `'\''` and the backslash `'\\'`, as well as certain whitespace characters like newline `'\n'` or tab `'\t'`.

String literals are given in double quotes (with special characters similarly escaped), for instance, `""`, `"first"`, `" and second"`. They can be concatenated with the `^` operator.[2]

```
# "" ^ "first" ^ " and second" ;;
- : string = "first and second"
```

[2] A useful trick is to use the escape sequence of a backslash, a newline, and any amount of whitespace, all of which will be ignored, so as to split a string over multiple lines. For instance,

```
# "First, " ^ "second, \
#             third, \
#             and fourth." ;;
- : string = "First, second, third, and fourth."
```

4.1.4 *Truth values and expressions*

There are two TRUTH VALUES, indicated in OCaml by the literals `true` and `false`. Logical reasoning based on truth values was codified by the British mathematician George Boole (1815–1864), leading to the use of the term *boolean* for such values, and the type name `bool` for them in OCaml.

Just as arithmetic values can be operated on with arithmetic operators, the truth values can be operated on with logical operators, such as operators for conjunction (&&), disjunction (||), and negation (`not`). (See Section B.3 for definitions of these operators.)

```
# false ;;
- : bool = false
# true || false ;;
- : bool = true
# true && false ;;
- : bool = false
# true && not false ;;
- : bool = true
```

The equality operator = tests two values[3] for equality, returning `true` if they are equal and `false` otherwise. There are other COMPARISON OPERATORS as well: < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), <> (not equal).

[3] This is the first example of a function that can apply to values of different types, a powerful idea that we will explore in detail in Chapter 9.

```
# 3 = 3 ;;
- : bool = true
# 3 > 4 ;;
- : bool = false
# 1 + 1 = 2 ;;
- : bool = true
# 3.1416 = 314.16 /. 100. ;;
- : bool = false
# true = false ;;
- : bool = false
# true = not false ;;
- : bool = true
# false < true ;;
- : bool = true
```

Exercise 9

Are any of the results of these comparisons surprising? See if you can figure out why the results are that way.

Of course, the paradigmatic use of truth values is in the ability to compute different values depending on the truth or falsity of a condition. The OCaml CONDITIONAL expression follows the template[4]

`if` $\langle expr_{test} \rangle$ `then` $\langle expr_{true} \rangle$ `else` $\langle expr_{false} \rangle$

whose effect is to return the value of the $\langle expr_{true} \rangle$ if the value of the test expression $\langle expr_{test} \rangle$ is `true` and the value of the $\langle expr_{false} \rangle$ if the value of $\langle expr_{test} \rangle$ is `false`.

```
# if 3 = 3 then 0 else 1 ;;
- : int = 0
# 2 * if 3 > 4 then 3 else 4 + 5 ;;
- : int = 18
# 2 * (if 3 > 4 then 3 else 4) + 5 ;;
- : int = 13
```

[4] We describe the syntax of the construct using the angle bracket convention for classes of expression used in BNF rules as introduced in Chapter 3. We will continue to do so throughout as we introduce new constructs of the language.

As mentioned in footnote 2 on page 34, in defining expression classes using this notation, we use subscripts to differentiate among different occurrences of the same expression class, as we have done here with the three instances of the $\langle expr \rangle$ class – $\langle expr_{test} \rangle$, $\langle expr_{true} \rangle$, and $\langle expr_{false} \rangle$.

4.2 OCaml expressions have types

We've introduced these additional values grouped according to their use. Integers are the type of things that integer operations are appropriate for; floating point numbers are the type of things that floating point operations are appropriate for; truth values are the type of things that logical operations are appropriate for. And conversely, it makes no sense to apply operations to values for which they are not appropriate. Therefore, OCaml is a TYPED language. Every expression of the language is associated with a type.

Figure 4.2: Small inconsistencies can lead to major problems: The explosion of the Ariane 5 on June 4, 1996.

Using values in ways inconsistent with their type is perilous. The maiden flight of the Ariane 5 rocket on June 4, 1996 ended spectacularly 37 seconds after launch when the rocket self-destructed. The reason? A floating-point value was used as an integer, causing an implicit conversion that overflowed. Using values in inappropriate ways

is a frequent source of bugs in code, even if not with the dramatic aftermath of the Ariane 5 explosion. As we will see, associating values with types can often prevent these kinds of bugs.

The OCaml language is STATICALLY TYPED, in that the type of an expression can be determined just by examining the expression in its context. It is not necessary to run the code in which an expression occurs in order to determine the type of an expression, as might be necessary in a DYNAMICALLY TYPED language (Python or JavaScript, for instance).

Types are themselves a powerful abstraction mechanism. Types are essentially *abstract values*. By reasoning about the types of expressions, we can convince ourselves of the correctness of code without having to run it.

Furthermore, OCaml is STRONGLY TYPED; values may not be used in ways inappropriate for their type. One of the ramifications of OCaml's strong typing is that functions only apply to values of certain types and only return values of certain types. For instance, the addition function specified by the + operator expects integer arguments and returns an integer result.

By virtue of strong, static typing, the programming system (compiler or interpreter) can tell the programmer when type constraints are violated *even before the program is run,* thereby preventing bugs before they happen. If you attempt to use a value in a manner inconsistent with its type, OCaml will complain with a typing error. For instance, integer multiplication can't be performed on floating point numbers or strings:

```
# 5 * 3 ;;
- : int = 15
# 5 * 3.1416 ;;
Line 1, characters 4-10:
1 | 5 * 3.1416 ;;
        ^^^^^^
Error: This expression has type float but an expression was
  expected of type
         int
# "five" * 3 ;;
Line 1, characters 0-6:
1 | "five" * 3 ;;
    ^^^^^^
Error: This expression has type string but an expression was
  expected of type
         int
```

Programmers using a language with strong static typing for the first time often find the frequent type errors limiting and even annoying. Furthermore, there are some computations that can't be expressed well with such strict limitations, especially low-level systems computations

Type	*Type expression*	*Example values*			*An example expression*
integers	`int`	`1`	`-2`	`42`	`(3 + 4) * 5`
floating point numbers	`float`	`3.14`	`-2.`	`2e12`	`(3.0 +. 4.) *. 5e0`
characters	`char`	`'a'`	`'&'`	`'\n'`	`char_of_int (int_of_char 's')`
strings	`string`	`"a"`	`"3 + 4"`		`"re" ^ "bus"`
truth values	`bool`	`true`	`false`		`true && not false`
unit	`unit`	`()`			`ignore (3 + 4)`

Table 4.1: Some of the atomic OCaml types with example values and an example expression.

that need access to the underlying memory representation of values. But a type error found at compile time is a warning that data use errors could show up at run time after the code has been deployed – and when it's far too late to repair it. Strong static type constraints are thus an example of a language restraint that frees programmers from verifying that their code does not contain "bad" operations by empowering the language interpreter to do so itself. (Looking ahead to the edict of prevention in Chapter 11, it makes the illegal inexpressible.)

4.2.1 *Type expressions and typings*

In OCaml, every type has a "name". These names are given as TYPE EXPRESSIONS, a kind of little language for naming types. Just as there are value expressions for specifying values, there are type expressions for specifying types.

In this language of type expressions, each ATOMIC TYPE has its own name. We've already seen the names of the integer, floating point, and truth value types – `int`, `float`, and `bool`, respectively – in the examples earlier in this chapter, because the REPL prints out a type expression for a value's type along with the value itself, for instance,

```
# 42 ;;
- : int = 42
# 3.1416 ;;
- : float = 3.1416
# false ;;
- : bool = false
```

Notice that the REPL presents the type of each computed value after a colon (`:`). (Why a colon? You'll see shortly.)

Table 4.1 provides a more complete list of some of the atomic types in OCaml (some not yet introduced), along with their type names, some example values, and an example expression that specifies a value of that type using some functions that return values of the given type. (We'll get to non-atomic (composite) types in Chapter 7.)

It is often useful to notate that a certain expression is of a certain

type. Such a TYPING is notated in OCaml using the `:` operator, placing the value to the left of the operator and its type to the right. So, for instance, the following typings hold:

- `42 : int`
- `true : bool`
- `3.14 *. 2. *. 2. : float`
- `if 3 > 4 then 3 else 4 : int`

The first states that the expression `42` specifies an integer value, the second that `true` specifies a boolean truth value, and so forth. The `:` operator is sometimes read as "the", thus "`42`, the integer" or "`true`, the bool". The typing operator is special in that it combines an expression from the value language (to its left) with an expression from the type language (to its right).

We can test out these typings right in the REPL. (The parentheses are necessary.)

```
# (42 : int) ;;
- : int = 42
# (true : bool) ;;
- : bool = true
# (3.14 *. 2. *. 2. : float) ;;
- : float = 12.56
# (if 3 > 4 then 3 else 4 : int) ;;
- : int = 4
```

The REPL generates an error when a value is claimed to be of an inappropriate type.

```
# (42 : float) ;;
Line 1, characters 1-3:
1 | (42 : float) ;;
     ^^
Error: This expression has type int but an expression was expected
  of type
         float
  Hint: Did you mean `42.'?
```

Exercise 10

Which of the following typings hold?

1. `3 + 5 : float`
2. `3. + 5. : float`
3. `3. +. 5. : float`
4. `3 : bool`
5. `3 || 5 : bool`
6. `3 || 5 : int`

Try typing these into the REPL to see what happens. (Remember to surround them with parentheses.)

Finally, in OCaml, expressions are IMPLICITLY TYPED. Although all expressions have types, and the types of expressions can be annotated using typings, *the programmer doesn't need to specify those types* in general. Rather, the OCaml interpreter can typically deduce the types of expressions at compile time using a process called TYPE INFERENCE. In fact, the examples shown so far depict this inference. The REPL prints not only the value calculated for each expression but also the type that it inferred for the expression.

4.3 *The unit type*

In OCaml, the phrases of the language are expressions, expressing values. In many other programming languages, the phrases of the language are not always used to express values. Rather, they are used as commands. They are of interest because of what they *do*, not what they *are*. This approach is especially prevalent in imperative programming, the term 'imperative' deriving from the Latin '*imperativus*', meaning 'pertaining to a command'. But OCaml, like other functional languages, is uniform in privileging expressions over commands.

Occasionally, we have an expression that really need compute no value. But since every expression has to have a value in OCaml, we need to assign a value to such expressions as well. In this case, we use the value `()`, spelled with an open and close parenthesis and pronounced "unit". This value is the only value of the type `unit`. Since the `unit` type has only one value, that value conveys no information, which is just what we want as the value of an expression whose value is irrelevant. The `unit` type will feature more prominently once we explore imperative programming within OCaml in Chapter 15.

Exercise 11

Give a typing for a value of the `unit` type.

4.4 *Functions are themselves values*

Functions play a central role in OCaml. They serve as the primary programming abstraction, as they do in many languages.

In a mathematical sense, a FUNCTION is simply a mapping from an input (called the function's ARGUMENT) to an output (the function's VALUE). Some functions that are built into OCaml are depicted in Figure 4.3.

OCaml is a FUNCTIONAL PROGRAMMING LANGUAGE, by which we mean more than that functions play a central role in the language. We

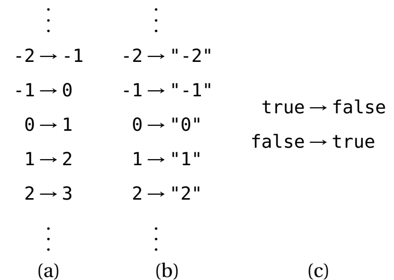

Figure 4.3: Three example functions: (a) the function from integers to their successors, available in OCaml as the `succ` function; (b) the function from integers to their string representation, available in OCaml as the `string_of_int` function; (c) the function mapping each boolean value onto its negation, available as the `not` function in OCaml.

mean that functions are FIRST-CLASS VALUES – they can be passed as arguments to functions or returned as the value of functions. Functions that take functions as arguments or return functions as values are referred to as HIGHER-ORDER FUNCTIONS, and the powerful programming paradigm that makes full use of this capability, which we will introduce in Chapter 8, is HIGHER-ORDER FUNCTIONAL PROGRAMMING.

Related to the idea that functions are values is that they have types as well. In Exercise 8, you used the `sqrt` function to take the square root of a floating point number. This function, `sqrt`, is itself a value and has a type. The type of a function expresses both the type of its argument (in this case, `float`) and the type of its output (again `float`). The type expression for a function (the type's "name") is formed by placing the symbol `->` (read "arrow" or "to") between the argument type and the output type. Thus the type for `sqrt` is `float -> float` (read "float arrow float" or "float to float"), or, expressed as a typing, `sqrt : float -> float`.

You can verify this typing yourself, just by evaluating `sqrt`:

```
# sqrt ;;
- : float -> float = <fun>
```

Since functions are themselves values, they can be evaluated, and the REPL performs type inference and provides the type `float -> float` along with a printed representation of the value itself `<fun>`, indicating that the value is a function of some sort.[5]

[5] The actual value of a function is a complex data object whose internal structure is not useful to print, so this abstract presentation `<fun>` is printed instead.

Because the argument type of `sqrt` is `float`, it can only be applied to values of that type. And since the result type of `sqrt` is `float`, only functions that take `float` arguments can apply to expressions like `sqrt 42.`.

Exercise 12

What are the types of the three functions – `succ`, `string_of_int`, and `not` – from Figure 4.3?

Exercise 13

Try applying the `sqrt` function to an argument of some type other than `float`, for instance, a value of type `bool`. What happens?

Of course, the real power in functional programming comes from defining your own functions. We'll move to this central topic in Chapter 6, but first, it is useful to provide a means of *naming* values (including functions), to which we turn in the next chapter.