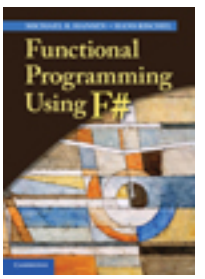


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

3 - Tuples, records and tagged values pp. 43-66

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.004

Cambridge University Press

3

Tuples, records and tagged values

Tuples, records and tagged values are compound values obtained by combining values of other types. Tuples are used in expressing "functions of several variables" where the argument is a tuple, and in expressing functions where the result is a tuple. The components in a record are identified by special identifiers called labels. Tagged values are used when we group together values of different kinds to form a single set of values. Tuples, records and tagged values are treated as "first-class citizens" in F#: They can enter into expressions and the value of an expression can be a tuple, a record or a tagged value. Functions on tuples, records or tagged values can be defined by use of patterns.

3.1 Tuples

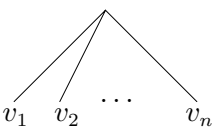
An ordered collection of n values $(v_1, v_2, \ldots, v_n)$, where $n > 1$, is called an *n-tuple*. Examples of n-tuples are:

```
(10, true);;
val it : int * bool = (10, true)

(("abc",1),-3);;
 val it : (string * int) * int = (("abc", 1), -3)
```

A 2-tuple like `(10,true)` is also called a *pair*. The last example shows that a pair, for example, `(("abc",1),-3)`, can have a component that is again a pair `("abc",1)`. In general, tuples can have arbitrary values as components. A 3-tuple is called a *triple* and a 4-tuple is called a *quadruple*. An expression like `(true)` is *not* a tuple but just the expression `true` enclosed in brackets, so there is *no* concept of 1-tuple. The symbol `()` denotes the only value of type `unit` (cf. Page 23).

The n-tuple $(v_1, v_2, \ldots, v_n)$ represents the graph:



The tuples `(true,"abc",1,-3)` and `((true,"abc"),1,-3)` contain the same values `true`, `"abc"`, `1` and `-3`, but they are different because they have a different structure. This difference is easily seen from the structure of the corresponding graphs in Figure 3.1, where the 4-tuple `(true,"abc",1,-3)` represents the graph with four branches

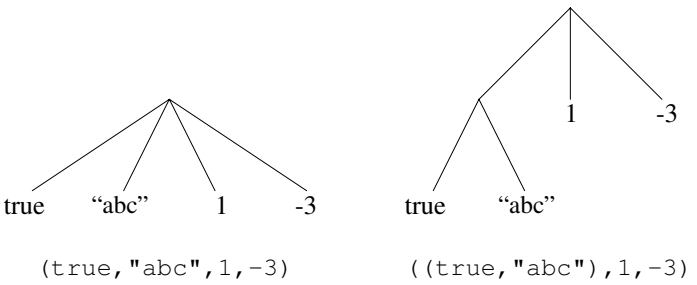


Figure 3.1 Graphs for tuple values

while the 3-tuple `((true,"abc"),1,-3)` represents the graph with three branches and a sub-graph with two branches.

Tuple expressions

A *tuple expression* $(expr_1, expr_2, \ldots, expr_n)$ is obtained by enclosing n expressions $expr_1$, $expr_2$, ..., $expr_n$ in parentheses. It has the type $\tau_1 * \tau_2 * \cdots * \tau_n$ when $expr_1$, $expr_2$, ..., $expr_n$ have types $\tau_1, \tau_2, \ldots, \tau_n$. For example:

`(1<2,"abc",1,1-4)`	has type	`bool * string * int * int`
`(true,"abc")`	has type	`bool * string`
`((2>1,"abc"),3-2,-3)`	has type	`(bool * string) * int * int`

Remark: The tuple type $\tau_1 * \tau_2 * \cdots * \tau_n$ corresponds to the *Cartesian Product*

$$A = A_1 \times A_2 \times \cdots \times A_n$$

of n sets $A_1, A_2, \ldots, A_n$ in mathematics. An element a of the set A is a tuple $a = (a_1, a_2, \ldots, a_n)$ of elements $a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$.

A tuple expression $(expr_1, expr_2, \ldots, expr_n)$ is *evaluated* from left to right, that is, by first evaluating $expr_1$, then $expr_2$, and so on. Tuple expressions can be used in declarations whereby an identifier is bound to a tuple value, for example:

```
let tp1 = ((1<2, "abc"), 1, 1-4);;
val tp1 : (bool * string) * int * int = ((true, "abc"), 1, -3)

let tp2 = (2>1, "abc", 3-2, -3);;
val tp2 : bool * string * int * int = (true, "abc", 1, -3)
```

Tuples are individual values

A tuple expression may contain identifiers that are already bound to tuples, for example:

```
let t1 = (true,"abc");;
val t1 : bool * string = (true, "abc")
```

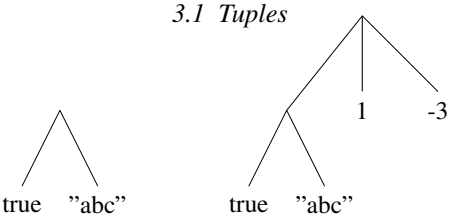


Figure 3.2 Graphs for tuples `(true,"abc")` and `((true, "abc"), 1, -3)`

```
let t2 = (t1,1,-3);;
val t2 : (bool * string) * int * int = ((true, "abc"), 1, -3)
```

The value bound to the identifier `t1` is then found as a subcomponent of the value bound to `t2` as shown in Figure 3.2. A fresh binding of `t1` is, however, *not* going to affect the value of `t2`:

```
let t1 = -7 > 2;;
val t1 : bool = false

t2;;
val it : (bool * string) * int * int = ((true, "abc"), 1, -3)
```

The subcomponent (true,"abc") is a value in its own right and it depends in no way on possible future bindings of `t1` once the value of the expression `(t1,1,-3)` has been evaluated.

Equality

Equality is defined for n-tuples of the same type, provided that equality is defined for the components. The equality is defined componentwise, that is, $(v_1, v_2, \ldots, v_n)$ is equal to $(v'_1, v'_2, \ldots, v'_n)$ if v_i is equal to v'_i for $1 \leq i \leq n$. This corresponds to equality of the graphs represented by the tuples. For example:

```
("abc", 2, 4, 9) = ("ABC", 2, 4, 9);;
val it : bool = false

(1, (2,true)) = (2-1, (2,2>1));;
val it : bool = true

(1, (2,true)) = (1, 2, 2>1);;
  (1, (2,true)) = (1, 2, 2>1);;
  ----------------^^^^^^^^^
stdin(25,18): error FS0001: Type mismatch. Expecting a
    int * (int * bool)
but given a
    int * (int * bool) * 'a
The tuples have differing lengths of 2 and 3
```

An error message occurs in the last example. The pair `(1, (2,true))` on the left-hand side of the equality has type `int * (int * bool)` while the tuple on the right-hand side has type `int * int * bool`. The system recognizes that these types are different and issues an error message.

Ordering

The *ordering* operators: >, >=, <, and <=, and the `compare` function are defined on n-tuples of the same type, provided ordering is defined for the components. Tuples are ordered lexicographically:

$$(x_1, x_2, \ldots, x_n) < (y_1, y_2, \ldots, y_n)$$

exactly when, for some k, where $1 \leq k \leq n$, we have:

$$x_1 = y_1 \wedge x_2 = y_2 \wedge \ldots \wedge x_{k-1} = y_{k-1} \wedge x_k < y_k$$

For example:

```
(1, "a") < (1, "ab");;
val it : bool = true

(2, "a") < (1, "ab");;
val it : bool = false
```

since "a" < "ab" holds while 2 < 1 does not.

The other comparison operators and the `compare` function can be defined in terms of = and < as usual, for example:

```
('a', ("b",true), 10.0) >= ('a', ("b",false), 0.0);;
val it : bool = true

compare ("abcd", (true, 1)) ("abcd", (false, 2));;
val it : int = 1
```

Tuple patterns

A tuple pattern represents a graph. For example, the pattern `(x,n)` is a *tuple pattern*. It represents the graph shown to the left containing the identifiers `x` and `n`:

The graph represented by the value `(3,2)` (shown to the right) *matches* the graph for the pattern in the sense that the graph for the value is obtained from the graph for the pattern by substituting suitable values for the identifiers in the pattern – in this case the value 3 for the identifier `x` and the value 2 for the identifier `n`. Hence, the pattern matching gives the bindings $\texttt{x} \mapsto 3$ and $\texttt{n} \mapsto 2$.

Patterns can be used on the left-hand side in a `let` declaration which binds the identifiers in the pattern to the values obtained by the pattern matching, for example:

```
let (x,n) = (3,2);;
val x : int = 3
val n : int = 2
```

Patterns may contain constants like the pattern `(x,0)`, for example, containing the constant `0`. It matches any pair (v_1, v_2) where $v_2 = 0$, and the binding $\mathtt{x} \mapsto v_1$ is then obtained:

```
let (x,0) = ((3,"a"),0);;
val x : int * string = (3, "a")
```

This example also illustrates that the pattern matching may bind an identifier (here: `x`) to a value which is a tuple.

The pattern `(x,0)` is *incomplete* in the sense that it just matches pairs where the second component is 0 and there are other pairs of type τ`*int` that do not match the pattern. The system gives a warning when an incomplete pattern is used:

```
 let (x,0) = ((3,"a"),0);;
  ----^^^^^
stdin(46,5): warning FS0025: Incomplete pattern matches on
this expression. For example, the value '(_,1)' may indicate a
case not covered by the pattern(s).
```

The warning can be ignored since the second component of `((3,"a"),0)` is, in fact, 0.

By contrast the declaration:

```
let (x,0) = (3,2);;
  let (x,0) = (3,2);;
  ----^^^^^
stdin(49,5): warning FS0025: Incomplete pattern matches on
this expression. For example, the value '(_,1)' may indicate a
case not covered by the pattern(s).
Microsoft.FSharp.Core.MatchFailureException: The match cases
   were incomplete at <StartupCode$FSI_0036>.$FSI_0036.main@()
Stopped due to error
```

generates an error message because the constant `0` in the pattern does not match the corresponding value `2` on the right-hand side. The system cannot generate any binding in this case.

The wildcard pattern can be used in tuple patterns. Every value matches this pattern, but the matching provides no bindings. For example:

```
let ((_,x),_,z) = ((1,true), (1,2,3), false);;
val z : bool = false
val x : bool = true
```

A pattern cannot contain multiple occurrences of the same identifier, so `(x,x)`, for example, is an illegal pattern:

```
let (x,x) = (1,1);;
   let (x,x) = (1,1);;
  -------^
 ... error FS0038: 'x' is bound twice in this pattern
```

3.2 Polymorphism

Consider the function `swap` interchanging the components of a pair:

```
let swap (x,y) = (y,x);;
val swap : 'a * 'b -> 'b * 'a

swap ('a',"ab");;
val it : string * char = ("ab", 'a')

swap ((1,3),("ab",true));;
val it : (string*bool) * (int*int) = (("ab", true), (1, 3))
```

The examples show that the function applies to all kinds of pairs. This is reflected in the type of the function: `'a * 'b -> 'b * 'a`.

The type of `swap` expresses that the argument (type `'a * 'b`) must be a pair, and that the value will be a pair (type `'b * 'a`) such that the first/second component of the value is of same type as the second/first component of the argument.

The type of `swap` contains two *type variables* `'a` and `'b`. A type containing type variables is called a *polymorphic type* and a function with polymorphic type like `swap` is called a *polymorphic function*. Polymorphic means "of many forms": In our case the F# compiler is able to generate a *single* F# function `swap` working on any kind of pairs and which is hence capable of handling data of many forms.

Polymorphism is related to overloading (cf. Section 2.5) as we in both cases can apply the same function name or operator to arguments of different types, but an overloaded operator denotes *different* F# functions for different argument types (like + denoting integer addition when applied to `int`'s and floating-point addition when applied to `float`'s).

There are two predefined, polymorphic functions

`fst: 'a * 'b -> 'a` and `snd: 'a * 'b -> 'b`

on pairs, that select the first and second component, respectively. For example:

```
fst((1,"a",true), "xyz");;
val it : int * string * bool = (1, "a", true)

snd('z', ("abc", 3.0));;
val it : string * float = ("abc", 3.0)
```

3.3 Example: Geometric vectors

A proper vector in the plane is a direction in the plane together with a non-negative length. The null vector is any direction together with the length 0. A vector can be *represented* by its set of Cartesian coordinates which is a pair of real numbers. A vector might instead be represented by its polar coordinates, which is also a pair of real numbers for the length and the angle. These two representations are different, and the operators on vectors (addition of vectors, multiplication by a scalar, etc.) are expressed by different functions on the representing pairs of numbers.

In the following we will just consider the Cartesian coordinate representation, where a vector in the plane will be represented by a value of type `float * float`.

We will consider the following operators on vectors:

Vector addition:	$(x_1, y_1) + (x_2, y_2)$	$=$	$(x_1 + x_2, y_1 + y_2)$
Vector reversal:	$-(x, y)$	$=$	$(-x, -y)$
Vector subtraction:	$(x_1, y_1) - (x_2, y_2)$	$=$	$(x_1 - x_2, y_1 - y_2)$
		$=$	$(x_1, y_1) + -(x_2, y_2)$
Multiplication by a scalar:	$\lambda\,(x_1, y_1)$	$=$	$(\lambda x_1, \lambda y_1)$
Dot product:	$(x_1, y_1) \cdot (x_2, y_2)$	$=$	$x_1 x_2 + y_1 y_2$
Norm (length):	$\|\|(x_1, y_1)\|\|$	$=$	$\sqrt{x_1^2 + y_1^2}$

We cannot use the operator symbols `+`, `-`, `*`, and so on, to denote the operations on vectors, as this would overwrite their initial meaning. But using `+.` for vector addition, `-.` for vector reversal and subtraction, `*.` for product by a scalar and `&.` for dot product, we obtain operators having a direct resemblance to the mathematical vector operators and having the associations and precedences that we would expect.

The prefix operator for vector reversal is declared by (cf. Section 2.9):

```
let (~-.) (x:float,y:float) = (-x,-y);;
val ( ~-. ) : float * float -> float * float
```

and the infix operators are declared by:

```
let (+.) (x1, y1) (x2,y2) = (x1+x2,y1+y2): float*float;;
val ( +. ) : float * float -> float * float -> float * float

let (-.) v1 v2 = v1 +. -. v2;;
val ( -. ) : float * float -> float * float -> float * float

let ( *.) x (x1,y1) = (x*x1, x*y1): float*float;;
val ( *. ) : float -> float * float -> float * float

let (&.) (x1,y1) (x2,y2) = x1*x2 + y1*y2: float;;
val ( &. ) : float * float -> float * float -> float
```

The norm function is declared using the `sqrt` function (cf. Table 2.5) by:

```
let norm(x1:float,y1:float) = sqrt(x1*x1+y1*y1);;
val norm : float * float -> float
```

These functions allow us to write vector expressions in a form resembling the mathematical notation for vectors. For example:

```
let a = (1.0,-2.0);;
val a : float * float = (1.0, -2.0)

let b = (3.0,4.0);;
val b : float * float = (3.0, 4.0)
```

```
let c = 2.0 *. a -. b;;
val c : float * float = (-1.0, -8.0)

let d = c &. a;;
val d : float = 15.0

let e = norm b;;
val e : float = 5.0
```

3.4 Records

A *record* is a generalized tuple where each component is identified by a *label* instead of the position in the tuple.

The record type must be declared before a record can be made. We may for example declare a type `person` as follows:

```
type Person = {age : int; birthday : int * int;
               name : string; sex : string};;
```

The keyword `type` indicates that this is a *type declaration* and the braces `{` and `}` indicate a record type. The (distinct) identifiers `age`, `birthday`, `name` and `sex` are called *record labels* and they are considered part of the type.

A value of type `Person` is entered as follows:

```
let john = {name =  "John"; age = 29;
            sex = "M"; birthday = (2,11)};;
val john : Person = {age = 29;
                     birthday = (2, 11);
                     name = "John";
                     sex = "M";}
```

This record contains the following *fields*: The string "`John`" with label `name`, the integer 29 with label `age`, the string "`M`" with label `sex`, and the integer pair (2,11) with label `birthday`.

The declaration creates the following binding of the identifier `john`:

$$\texttt{john} \mapsto \{\, \texttt{age} \mapsto 29\,,\, \texttt{birthday} \mapsto (2{,}11)\,,\, \texttt{name} \mapsto \text{“}\texttt{John}\text{”}\,,\, \texttt{sex} \mapsto \text{“}\texttt{M}\text{”}\,\}$$

A record is hence a *local environment* packaged in a certain way. It contains a binding of each record label to the corresponding value.

A field in the record denoted by `john` is obtained by suffixing the identifier `john` with the corresponding record label:

```
john.birthday;;
val it : int * int = (2, 11)
john.sex;;
val it : string = "M"
```

Equality and ordering

The equality of two records with the same type is defined componentwise from the equality of values associated with the same labels, so the ordering of the components in the record is of no importance when entering values. For example

```
john = {age = 29; name = "John";
        sex = "M"; birthday = (2,11)};;
val it : bool = true
```

Hence two records are equal if they are of the same type and contain the same local bindings of the labels.

Ordering of records is based on a lexicographical ordering using the ordering of the labels in the record type declaration. Consider, for example:

```
type T1 = {a:int; b:string};;
let v1 = {a=1; b="abc"};;
let v2 = {a=2; b="ab"};;
v1<v2;;
val it : bool = true

type T2 = {b:string; a:int};;
let v1' = {T2.a=1; b="abc"};;
let v2' = {T2.a=2; b="ab"};;
v1'>v2';;
val it : bool = true
```

The value `v1` is smaller than the value `v2` because the label `a` occurs first in the record type `T1` and `v1.a = 1` is smaller than `v2.a = 2` – while the value `v1'` is larger than the value `v2'` because the label `b` occurs first in the record type `T2` and `v1'.b = "abc"` is larger than `v2'.b = "ab"`.

The composite identifier `T2.a` consists of the record label `a` prefixed with the record type `T2`. It is used in order to resolve the ambiguity created by reuse of record labels.

Note that the values `v1` and `v1'` cannot be compared as they are of different types.

Record patterns

A *record pattern* is used to decompose a record into its fields. The pattern

```
{name = x; age = y; sex = s; birthday =(d,m)}
```

denotes the graph shown in Figure 3.3. It generates bindings of the identifiers `x`, `y`, `s`, `d` and `m` when matched with a person record:

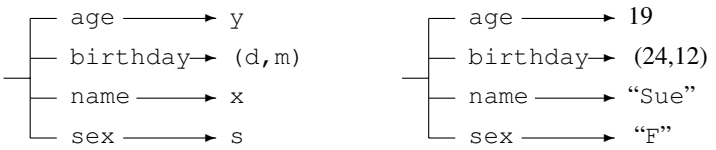


Figure 3.3 Record pattern and record

```
let sue = {name="Sue"; age = 19; sex="F";
           birthday = (24,12)};;
let {name = x; age = y; sex = s; birthday = (d,m)} = sue;;
val y : int = 19
val x : string = "Sue"
val s : string = "F"
val m : int = 12
val d : int = 24
```

Record patterns are used when defining functions. Consider, for example, the declaration of a function `age` where the argument is a record of type `Person`:

```
let age {age = a; name = _; sex=_; birthday=_} = a;;
val age : Person -> int

let isYoungLady {age=a; sex=s; name=_; birthday=_}
                                          = a < 25 && s = "F";;
val isYoungLady : Person -> bool

age john;;
val it : int = 29

isYoungLady john;;
val it : bool = false

isYoungLady sue;;
val it : bool = true
```

The type of the above functions can be inferred from the context since `name`, `age`, and so on are labels of the record type `Person` only.

3.5 Example: Quadratic equations

In this section we consider the problem of finding solutions to quadratic equations

$$ax^2 + bx + c = 0$$

with real coefficients a, b, c.

The equation has no solution in the real numbers if the *discriminant* $b^2 - 4ac$ is negative; otherwise, if $b^2 - 4ac \geq 0$ and $a \neq 0$, then the equation has the solutions x_1 and x_2 where:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Note that $x_1 = x_2$ if $b^2 - 4ac = 0$.

We may represent the equation $ax^2+bx+c = 0$ by the triple (a, b, c) of real numbers and the solutions x_1 and x_2 by the pair (x_1, x_2) of real numbers. This representation is captured in the type declarations:

```
type Equation = float * float * float;;
type Equation = float * float * float
```

```
type Solution = float * float;;
type Solution = float * float
```

A function:

```
solve: Equation -> Solution
```

for computing the solutions of the equation should then have the indicated type. Note that type declarations like the ones above are useful in program documentation as they communicate the intention of the program in a succinct way. The system does, however, just treat the identifiers `Equation` and `Solution` as shorthand for the corresponding types.

Error handling

The function `solve` must give an error message when $b^2 - 4ac < 0$ or $a = 0$ as there is no solution in these cases. Such an error message can be signalled by using an *exception*. An exception is named by an *exception declaration*. We may, for example, name an exception `Solve` by the declaration:

```
exception Solve;;
exception Solve
```

The declaration of the function `solve` is:

```
let solve(a,b,c) =
     if b*b-4.0*a*c < 0.0 || a = 0.0 then raise Solve
     else ((-b + sqrt(b*b-4.0*a*c))/(2.0*a),
           (-b - sqrt(b*b-4.0*a*c))/(2.0*a));;
val solve : float * float * float -> float * float
```

The `then` branch of this declaration contains the expression: `raise Solve`. An evaluation of this expression terminates with an error message. For example:

```
solve(1.0, 0.0, 1.0);;
FSI_0015+Solve: Exception of type 'FSI_0015+Solve' was thrown.
   at FSI_0016.solve(Double a, Double b, Double c)
   at <StartupCode$FSI_0017>.$FSI_0017.main@()
Stopped due to error
```

We say that the exception `Solve` is *raised*. Note that the use of the exception does not influence the type of `solve`.

Other examples of the use of `solve` are:

```
solve(1.0, 1.0, -2.0);;
val it : float * float = (1.0, -2.0)

solve(2.0, 8.0, 8.0);;
val it : float * float = (-2.0, -2.0)
```

An alternative to declaring your own exception is to use the built-in function:

```
failwith: string -> 'a
```

that takes a string as argument. An application `failwith` *s* raises the exception `Faiure` *s* and the argument string *s* is shown on the console. The function can be applied in any context because the function value has a polymorphic type. For example:

```
let solve(a,b,c) =
    if b*b-4.0*a*c < 0.0 || a = 0.0
    then failwith "discriminant is negative or a=0.0"
    else ((-b + sqrt(b*b-4.0*a*c))/(2.0*a),
          (-b - sqrt(b*b-4.0*a*c))/(2.0*a));;

solve(0.0,1.0,2.0);;
System.Exception: discriminant is negative or a=0.0
   at FSI_0037.solve(Double a, Double b, Double c)
   at <StartupCode$FSI_0038>.$FSI_0038.main@()
Stopped due to error
```

We shall on Page 63 study how raised exceptions can be caught.

3.6 Locally declared identifiers

It is often convenient to use *locally declared* identifiers in function declarations. Consider for example the above declaration of the function `solve`. The expression `b*b-4.0*a*c` is evaluated three times during the evaluation of `solve(1.0,1.0,-2.0)` and this is not satisfactory from an efficiency point of view. Furthermore, the readability of the declaration suffers from the repeated occurrences of the same subexpression.

These problems are avoided using a locally declared identifier `d`:

```
let solve(a,b,c) =
    let d = b*b-4.0*a*c
    if d < 0.0 || a = 0.0
    then failwith "discriminant is negative or a=0.0"
    else ((-b + sqrt d)/(2.0*a),(-b - sqrt d)/(2.0*a));;
val solve : float * float * float -> float * float
```

There is room for improvement in the above declaration as the expression `sqrt d` is evaluated twice during the evaluation of a function value. This leads to yet another declaration of `solve` with a further locally declared identifier `sqrtD`:

```
let solve(a,b,c) =
    let sqrtD =
        let d = b*b-4.0*a*c
        if d < 0.0 || a = 0.0
        then failwith "discriminant is negative or a=0.0"
        else sqrt d
    ((-b + sqrtD)/(2.0*a),(-b - sqrtD)/(2.0*a));;
 val solve : float * float * float -> float * float
```

The evaluation of `solve(1.0,1.0,-2.0)` proceeds as follows, where *env* denotes the environment:

$$env = [\texttt{a} \mapsto 1.0,\ \texttt{b} \mapsto 1.0,\ \texttt{c} \mapsto -2.0]$$

obtained by binding the parameters `a`, `b` and `c` to the actual values 1.0, 1.0 and -2.0:

	Expression	Environment	Note
	`solve(1.0,1.0,-2.0)`		
$\rightsquigarrow$	`let sqrtD = ...,`	*env*	(1)
$\rightsquigarrow$	Start evaluating subexpression		
	`let d=b*b-4.0...,`	*env*	(2)
	$\rightsquigarrow$ `if d < 0...,`	*env* plus `d` $\mapsto 9.0$	(3)
	$\rightsquigarrow$ 3.0	*env* plus `d` $\mapsto 9.0$	(4)
	End evaluating subexpression		(5)
$\rightsquigarrow$	`((-b + sqrtD...`	*env* plus `sqrtD` $\mapsto 3.0$	(6)
$\rightsquigarrow$	$(1.0, -2.0)$		(7)

1. The binding of `sqrtD` can only be established when the value of the subexpression has been evaluated.
2. The evaluation of the subexpression starts with the declaration `let b = ...`.
3. The expression `b*b-4.0*a*c` evaluates to 9.0 using the bindings in *env*. A binding of `d` to this value is added to the environment.
4. The evaluation of `if d < 0.0 ... else sqrt d` gives the value 3.0 using the bindings in the environment *env* plus `d` $\mapsto 9.0$.
5. The evaluation of the subexpression is completed and the binding of `d` is removed from the environment.
6. A binding of `sqrtD` to the value 3.0 is added to the environment, and the expression `((-b + sqrtD ...` is evaluated in this environment.
7. The bindings of `a`, `b`, `c` and `sqrtD` are removed and the evaluation terminates with result $(1.0, -2.0)$.

Note the role of *indentation* in F#. The `let`-expression:

```
let d = b*b-4.0*a*c
if d < 0.0 || a = 0.0
then failwith "discriminant is negative or a=0.0"
else sqrt d
```

is terminated by the occurrence of a less indented line:

```
((-b + sqrtD)/(2.0*a),(-b - sqrtD)/(2.0*a));;
```

and this also ends the lifetime of the binding of `d`. One says that the `let`-expression constitutes the *scope* of the declaration of `d`.

The surrounding `let`-expression

```
let sqrtD =
    let d = b*b-4.0*a*c
    if d < 0.0 || a = 0.0
    then failwith "discriminant is negative or a=0.0"
    else sqrt d
((-b + sqrtD)/(2.0*a),(-b - sqrtD)/(2.0*a))
```

is terminated by the double semicolon. Note that the expression `((-b + ...` must be on the same indentation level as `let sqrtD =`.

A `let`-expression may contain more than one local declaration as shown in yet another version of `solve` (probably the most readable):

```
let solve(a,b,c) =
    let d = b*b-4.0*a*c
    if d < 0.0 || a = 0.0
    then failwith "discriminant is negative or a=0.0"
    else let sqrtD = sqrt d
         ((-b + sqrtD)/(2.0*a),(-b - sqrtD)/(2.0*a));;
val solve : float * float * float -> float * float
```

The evaluation of `solve(1.0,1.0,-2.0)` in this version of the function will add the binding of `d` to the environment *env*. Later the binding of `sqrtD` is further added without removing the binding of `b`. Finally the expression in the last line is evaluated and the bindings of `a`, `b`, `c`, `d` and `sqrtD` are all removed at the same time.

3.7 Example: Rational numbers. Invariants

A *rational* number q is a fraction $q = \frac{a}{b}$, where a and b are integers with $b \neq 0$.

Ideas to express the operations on rational numbers by function declarations come easily from the following well-known rules of arithmetic, where a, b, c and d are integers such that $b \neq 0$ and $d \neq 0$:

$$\begin{aligned}
\frac{a}{b} + \frac{c}{d} &= \frac{ad+bc}{bd} \\
\frac{a}{b} - \frac{c}{d} &= \frac{a}{b} + \frac{-c}{d} = \frac{ad-bc}{bd} \\
\frac{a}{b} \cdot \frac{c}{d} &= \frac{ac}{bd} \\
\frac{a}{b} \,/\, \frac{c}{d} &= \frac{a}{b} \cdot \frac{d}{c} \quad \text{where} \quad c \neq 0 \\
\frac{a}{b} = \frac{c}{d} &\quad \text{exactly when} \quad ad = bc
\end{aligned} \tag{3.1}$$

Representation. Invariant

We use the *representation* (a, b), where $b > 0$ and where the fraction $\frac{a}{b}$ is irreducible, that is, $\texttt{gcd}(a, b) = 1$, to represent the rational number $\frac{a}{b}$. Thus, a value `(a,b)` of type `int * int` represents a rational number if $b > 0$ and $\texttt{gcd}(a, b) = 1$, and we name this condition the *invariant* for pairs representing rational numbers. Any rational number has a unique *normal form* of this kind. This leads to the type declaration:

```
type Qnum = int*int;;    // (a,b) where b > 0 and gcd(a,b) = 1
```

where the invariant is stated as a comment to the declaration. (The declaration of `gcd` is found on Page 15.)

Operators

It is convenient to declare a function `canc` that cancels common divisors and thereby reduces any fraction with non-zero denominator to the normal form satisfying the invariant:

```
let canc(p,q) =
   let sign = if p*q < 0 then -1 else 1
   let ap = abs p
   let aq = abs q
   let d  = gcd(ap,aq)
   (sign * (ap / d), aq / d);;
```

In the below declarations for the other functions, `canc` is applied to guarantee that the resulting values satisfy the invariant.

When a rational number is generated from a pair of integers, we must check for division by zero and enforce that the invariant is established for the result. The function `mkQ` does that by the use of `canc`:

```
let mkQ = function
   | (_,0)  -> failwith "Division by zero"
   | pr     -> canc pr;;
```

The operators on rational numbers are declared below. These declarations follow the rules (3.1) for rational numbers. We assume that the arguments are legal representations of rational numbers, that is, they respect the invariant. Under this assumption, the result of any of the functions must respect the invariant. This is enforced by the use of `canc` and `mkQ`:

```
let (.+.) (a,b) (c,d) = canc(a*d + b*c, b*d);;      // Addition

let (.-.) (a,b) (c,d) = canc(a*d - b*c, b*d);;  // Subtraction

let (.*.) (a,b) (c,d) = canc(a*c, b*d);;      // Multiplication

let (./.) (a,b) (c,d) = (a,b) .*. mkQ(d,c);;        // Division

let (.=.) (a,b) (c,d) = (a,b) = (c,d);;             // Equality
```

Note that the definition of equality assumes the invariant. Equality should be declared by `a*d=b*c` if we allow integer pairs *not* satisfying the invariant as there would then be *many different* integer pairs representing the *same* rational number.

It is straightforward to convert a rational number representation to a string:

```
let toString(p:int,q:int) = (string p) + "/" + (string q);;
```

as the representation is unique. We can operate on rational numbers in a familiar manner:

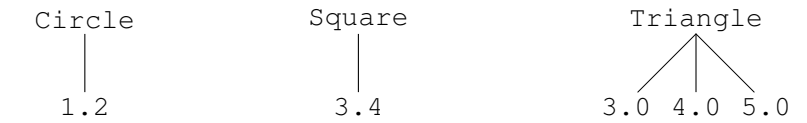
```
let q1 = mkQ(2,-3);;
val q1 : int * int = (-2, 3)
let q2 = mkQ(5,10);;
val q2 : int * int = (1, 2)
let q3 = q1 .+. q2;;
val q3 : int * int = (-1, 6)

toString(q1 .-. q3 ./. q2);;
val it : string = "-1/3"
```

3.8 Tagged values. Constructors

Tagged values are used when we group together values of different kinds to form a single set of values.

For example, we may represent a circle by its radius r, a square by its side length a, and a triangle by the triple (a, b, c) of its side lengths a, b and c. Circles, squares, and triangles may then be grouped together to form a *single* collection of *shapes* if we put a *tag* on each representing value. The tag should be `Circle`, `Square`, or `Triangle` depending on the kind of shape. The circle with radius `1.2`, the square with side length `3.4` and the triangle with side lengths `3.0`, `4.0` and `5.0` are then represented by the tagged values shown in the following graphs:



`Circle 1.2` `Square 3.4` `Triangle(3.0,4.0,5.0)`

In F#, a collection of tagged values is declared by a *type* declaration. For example, a type for shapes is declared by:

```
type Shape = | Circle of float
             | Square of float
             | Triangle of float*float*float;;
type Shape =
  | Circle of float
  | Square of float
  | Triangle of float * float * float
```

Constructors and values

The response from F# indicates that `Shape` names a type, and that `Circle`, `Square` and `Triangle` are bound to *value constructors*. These value constructors are functions and they give a *tagged value* of type `Shape` when applied to an argument.

For example, `Circle` is a value constructor with type `float -> Shape`. This means that `Circle` r denotes a value of type `Shape`, for every float r. For example, `Circle 1.2` denotes the leftmost graph in the above figure and `Circle 1.2` is an example of a *tagged* value, where `Circle` is the tag.

We can observe that `Circle 1.2` is a value which is not evaluated further by F#:

```
Circle 1.2;;
val it : Shape = Circle 1.2
```

as the value in the answer is equal to the expression being evaluated, that is, `Circle 1.2`.

Values can be constructed using `Square` and `Triangle` in a similar way.

Since constructors are functions in F#, `Circle` can be applied to an expression of type `float`:

```
Circle(8.0 - 2.0*3.4);;
val it : Shape = Circle 1.2
```

Thus, the declaration of `Shape` allows one to write tagged values like `Circle 1.2`, `Square 3.4` and `Triangle(3.0,4.0,5.0)` using the constructors, and any value of type `Shape` has one of the forms:

`Circle` r
`Square` a
`Triangle` (a, b, c)

for some float value r, float value a, or triple (a, b, c) of float values.

Equality and ordering

Equality and ordering are defined for tagged values provided they are defined for their components. Two tagged values are equal if they have the same constructor and their components are equal. This corresponds to equality of the graphs represented by the tagged values. For example:

```
Circle 1.2 = Circle(1.0 + 0.2);;
val it : bool = true

Circle 1.2 = Square 1.2;;
val it : bool = false
```

The sequence in which the tags occur in the declaration is significant for the ordering. For example, any circle is smaller than any square, which again is smaller than any triangle due to the order in which the corresponding tags are declared. For example:

```
Circle 1.2 < Circle 1.0;;
val it : bool = false
```

```
Circle 1.2 < Square 1.2;;
val it : bool = true

Triangle(1.0,1.0,1.0) > Square 4.0;;
val it : bool = true
```

Constructors in patterns

Constructors can be used in patterns. For example, an area function for shapes is declared by:

```
let area = function
    | Circle r        -> System.Math.PI * r * r
    | Square a        -> a * a
    | Triangle(a,b,c) ->
          let s = (a + b + c)/2.0
          sqrt(s*(s-a)*(s-b)*(s-c));;
val area : Shape -> float
```

The pattern matching treats constructors differently from other identifiers:

> A constructor matches itself only in a pattern match while other identifiers match any value.

For example, the value `Circle 1.2` will match the pattern `Circle r`, but not the other patterns in the function declaration. The matching binds the identifier `r` to the value `1.2`, and the expression `Math.pi * r * r` is evaluated using this binding:

$$\begin{aligned} & \texttt{area (Circle 1.2)} \\ \rightsquigarrow\ & (\texttt{Math.PI * r * r}, [\texttt{r} \mapsto \texttt{1.2}]) \\ \rightsquigarrow\ & \texttt{...} \end{aligned}$$

The value `Triangle(3.0,4.0,5.0)` will in a similar way only match the pattern in the third clause in the declaration, and we get bindings of `a`, `b` and `c` to `3.0`, `4.0` and `5.0`, and the `let` expression is evaluated using these bindings:

$$\begin{aligned} & \texttt{area (Triangle(3.0,4.0,5.0))} \\ \rightsquigarrow\ & (\texttt{let s = ...}, [\texttt{a} \mapsto \texttt{3.0}, \texttt{b} \mapsto \texttt{4.0}, \texttt{c} \mapsto \texttt{5.0}]) \\ \rightsquigarrow\ & \texttt{...} \end{aligned}$$

Invariant for the representation of shapes

Some values of type `Shape` do not represent geometric shapes. For example, `Circle -1.0` does not represent a circle, as a circle cannot have a negative radius, `Square -2.0` does not represent a square, as a square cannot have a negative side length, and

```
Triangle(3.0, 4.0, 7.5)
```

does not represent a triangle, as $7.5 > 3.0 + 4.0$ and, therefore, one of the triangle inequalities is not satisfied.

Therefore, there is an *invariant* for this representation of shapes: the real numbers have to be positive, and the triangle inequalities must be satisfied. This invariant can be declared as a predicate `isShape`:

```
let isShape = function
    | Circle r        -> r > 0.0
    | Square a        -> a > 0.0
    | Triangle(a,b,c) ->
        a > 0.0 && b > 0.0 && c > 0.0
        && a < b + c && b < c + a && c < a + b;;
val isShape : Shape -> bool
```

We consider now the declaration of an area function for geometric shapes that raises an exception when the argument of the function does not satisfy the invariant. If we try to modify the above area function:

```
let area x = if not (isShape x)
             then failwith "not a legal shape"
             else ...
```

then the `else`-branch must have means to select the right area-expression depending on the form of `x`. This is done using a `match` ... `with` ... expression:

```
let area x =
    if not (isShape x)
    then failwith "not a legal shape" raise
    else match x with
         | Circle r        -> System.Math.PI * r * r
         | Square a        -> a * a
         | Triangle(a,b,c) ->
             let s = (a + b + c)/2.0
             sqrt(s*(s-a)*(s-b)*(s-c));;
val area : Shape -> float
```

The modified `area` function computes the area of legal values of the type `Shape` and terminates the evaluation raising an exception for illegal values:

```
area (Triangle(3.0,4.0,5.0));;
val it : float = 6.0

area (Triangle(3.0,4.0,7.5));;
System.Exception: not a legal shape
 ...
```

3.9 Enumeration types

Value constructors need not have any argument, so we can make special type declarations like:

```
type Colour = Red | Blue | Green | Yellow | Purple;;
type Colour =
  | Red
  | Blue
  | Green
  | Yellow
  | Purple
```

Types like `Colour` are called *enumeration types*, as the declaration of `Colour` just enumerates five constructors:

```
Red, Blue, Green, Yellow, Purple
```

where each constructor is a value of type `Colour`, for example:

```
Green;;
val it : Colour = Green
```

Functions on enumeration types may be declared by pattern matching:

```
let niceColour = function
    | Red  -> true
    | Blue -> true
    | _    -> false;;
val niceColour : Colour -> bool

niceColour Purple;;
val it : bool = false
```

The days in a month example on Page 4 can be nicely expressed using an enumeration type:

```
type Month = January | February | March | April
             | May | June | July | August | September
             | October | November | December;;

let daysOfMonth = function
    | February                          -> 28
    | April | June | September | November -> 30
    | _                                 -> 31;;
```

The Boolean type is actually a predefined enumeration type:

```
type bool = false | true
```

where the order of the constructors reflects that `false` < `true`. Notice that user-defined constructors must start with uppercase letters.

3.10 Exceptions

Exceptions have already been used in several examples earlier in this chapter. In this section we give a systematic account of this subject.

Raising an exception terminates the evaluation of a call of a function as we have seen for the `solve` function on Page 53 that raises the exception `Solve` when an error situation is encountered. In the examples presented so far the exception propagates all the way to top level where an error message is issued.

It is possible to *catch* an exception using a `try...with` expression as in the following `solveText` function:

```
let solveText eq =
    try
        string(solve eq)
    with
    | Solve -> "No solutions";;
val solveText : float * float * float -> string
```

It calls `solve` with a float triple `eq` representing a quadratic equation and returns the string representation of the solutions of the equation:

```
solveText (1.0,1.0,-2.0);;
val it : string = "(1, -2)"
```

The string "`No solutions`" is returned if the equation has no solutions:

```
solveText (1.0, 0.0, 1.0);;
val it : string = "No solutions"
```

An application of the function `failwith` s will raise the exception `Failure` s and this exception can also be caught. Application of the function `mkQ` (see Page 57), for example, will call `failwith` in the case of a division by zero:

```
try
    toString(mkQ(2,0))
with
| Failure s -> s;;
val it : string = "Division by zero"
```

A `try...with` expression has the general form:

`try` e `with` *match*

where e is an expression (possibly extending over several lines) and *match* is a construct of the form:

| pat_1 -> e_1
| pat_2 -> e_2
...
| pat_n -> e_n

with patterns $pat_1, \ldots pat_n$ and expressions $e_1, \ldots, e_n$.

A `try` e `with` $match$ expression is evaluated as follows:

- Evaluate the expression e. If this evaluation terminates normally with a value v then return v as the result of evaluating the `try ... with ...` expression.
- If the evaluation raises an exception *Exc* then evaluate *match* by matching *Exc* to the patterns $pat_1, \ldots pat_n$. If *Exc* matches a pattern pat_k then evaluate the corresponding expression e_k. If *Exc* matches none of the patterns then propagate the exception as a result of evaluating the `try ... with ...` expression.

The exception mechanism in F# and .NET is *not* intended for use in the "normal case" in a program but for *error handling only*.

Library functions (e.g., for performing I/O) may raise exceptions that can only be captured using a *match on type* (cf. Section 7.7).

3.11 Partial functions. The option type

A function f is a *partial* function on a set A if the domain of f is a proper subset of A. For example, the factorial function is a partial function on the set of integers because it is undefined on negative integers.

In declaring a partial function, F# offers the programmer three ways of handling argument values where the function is *undefined*:

1. The evaluation of the function value does not terminate.
2. The evaluation of the function value is terminated by raising an exception.
3. The evaluation of the function value gives a special result, indicating that the function is undefined for the actual argument.

The first choice was used in the declaration of the factorial function `fact`, where, for example, the evaluation of `fact -1` never terminates.

The second choice was selected for the improved `area` function (cf. Page 61).

The third choice uses the predefined `option` type:

```
type 'a option = None | Some of 'a
```

where `None` is used as result for arguments where the function is undefined while `Some` v is used when the function has value v.

The constructor `Some` is polymorphic and can be applied to values of any type:

```
Some false;;
val it : bool option = Some false

Some (1, "a");;
val it : (int * string) option = Some (1, "a")
```

The value `None` is a polymorphic value of type `'a option`:

```
None;;
val it : 'a option = None
```

The library function

```
Option.get : 'a option -> 'a
```

"removes the `Some`", that is, `Option.get(Some` n`)` $= n$. It raises an exception when applied to `None`. For example:

```
Option.get(Some (1,"a"));;
val it : int * string = (1, "a")

Option.get(Some 1);;
val it : int = 1

Option.get None + 1;;
System.ArgumentException: The option value was None ...
```

We may, for instance, declare a modified factorial function `optFact`(n) with value `Some` $n!$ for $n \geq 0$ and `None` for $n < 0$:

```
let optFact n = if n < 0 then None else Some(fact n);;
val optFact : int -> int option
```

The function application `optFact` n always gives a result:

```
optFact 5;;
val it : int option = Some 120

optFact -2;;
val it : int option = None
```

The declaration of `optFact` presumes that `fact` has already been declared. An independent declaration of `optFact` is achieved using the `Option.get` function:

```
let rec optFact = function
  | 0             -> Some 1
  | n when n > 0 -> Some(n * Option.get(optFact(n-1)))
  | _             -> None;;
val optFact : int -> int option
```

Note the use of guarded patterns in this declaration (cf. Section 2.10).

Summary

This chapter introduces the notions of tuples and tuple types, the notions of records and record types, and the notions of tagged values and tagged-value types. Tuples and records are composite values, and we have introduced the notion of patterns that is used to decompose a composite value into its parts. Tagged values are used to express disjoint unions.

An operator can be given infix mode and precedence, and this feature was exploited in writing the operators on geometric vectors in the same way as they are written in mathematical notation.

The notion of exceptions was introduced for handling errors and `let` expressions were introduced for having locally declared identifiers.

Exercises

3.1 A time of day can be represented as a triple $(hours, minutes, f)$ where f is either AM or PM – or as a record. Declare a function to test whether one time of day comes before another. For example, `(11,59,"AM")` comes before `(1,15,"PM")`. Make solutions with triples as well as with records. Declare the functions in infix notation.

3.2 The former British currency had 12 pence to a shilling and 20 shillings to a pound. Declare functions to add and subtract two amounts, represented by triples $(pounds, shillings, pence)$ of integers, and declare the functions when a representation by records is used. Declare the functions in infix notation with proper precedences, and use patterns to obtain readable declarations.

3.3 The set of *complex numbers* is the set of pairs of real numbers. Complex numbers behave almost like real numbers if addition and multiplication are defined by:

$$\begin{aligned}(a,b)+(c,d) &= (a+c, b+d)\\ (a,b)\cdot(c,d) &= (ac-bd, bc+ad)\end{aligned}$$

1. Declare suitable infix functions for addition and multiplication of complex numbers.
2. The inverse of (a,b) with regard to addition, that is, $-(a,b)$, is $(-a,-b)$, and the inverse of (a,b) with regard to multiplication, that is, $1/(a,b)$, is $(a/(a^2+b^2), -b/(a^2+b^2))$ (provided that a and b are not both zero). Declare infix functions for subtraction and division of complex numbers.
3. Use `let`-expressions in the declaration of the division of complex numbers in order to avoid repeated evaluation of identical subexpressions.

3.4 A straight line $y = ax + b$ in the plane can be represented by the pair (a,b) of real numbers.

1. Declare a type `StraightLine` for straight lines.
2. Declare functions to mirror straight lines around the x and y-axes.
3. Declare a function to give a string representation for the equation of a straight line.

3.5 Make a type `Solution` capturing the three capabilities for roots in a quadratic equation: two roots, one root and no root (cf. Section 3.5). Declare a corresponding `solve` function.

3.6 Solve Exercise 3.1 using tagged values to represent AM and PM.

3.7 Give a declaration for the area function on Page 61 using guarded patterns rather than an `if...then...else` expression.