Modules and Data Type Declarations

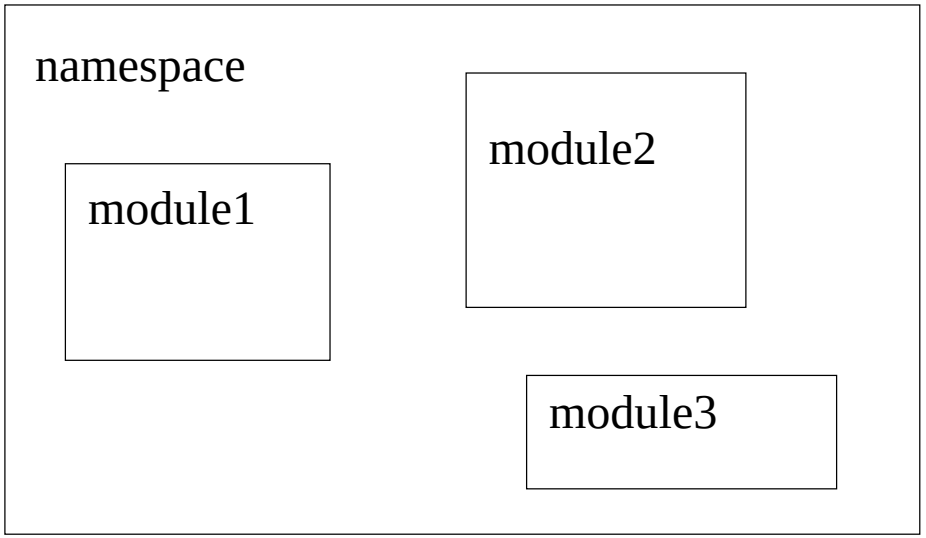
Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

Modules and Data Types

F# code is packaged in *name spaces*, and *modules*

We'll not talk much about namespaces now: basically, a namespace can contain a number of modules:



Modules

A module contains a number of declarations

The *scope* of the declarations is the module: this is where they are visible

Module $=$ software component containing functions, data types, . . .

Good for packaging libraries to be reused in other F# programs

Declaring Modules

Syntax of modules:

```
module xxx
  ...declarations...
```

Example:

```
module Allan
let f x = x + 17
let g x = f x + f (x*2)
```

How access `f` and `g`?

```
module Allan
let f x = x + 17
let g x = f x + f (x*2) // f is visible here
```

Inside `Allan`: `f` and `g` visible (the module is their *scope*)

Outside `Allan` (in same name space): `f` and `g` visible *if prefixed* with their module name

```
Allan.f 4711
```

Opening Modules

A module can be *opened* to make its declared identifiers visible without the prefix:

```
module Allan
let f x = x + 17
let g x = f x + f (x*2) // f is visible here

open Allan

let h x = f (x + 3) - g x
```

Order of Declarations

In F#, the order of declarations matter:

```
let f x = x + 17
let g x = f x + f (x*2) // OK, f is visible here

let g x = f x + f (x*2) // Not OK, f is not visible here
let f x = x + 17
```

So, an entity is really not in scope in its module until after it has been declared

A Simple Module Example

A simple module `Vector` with our previous vector operations on tuples:

```
module Vector
let vAdd (x1,y1) (x2,y2) = (x1+x2,y1+y2) : float * float
let vSub (x1,y1) (x2,y2) = (x1-x2,y1-y2) : float * float
let vLen (x,y) = sqrt (x**2.0 + y**2.0)
```

In the same namespace, the `Vector` module can now be opened:

```
open Vector

let v1 = (1.0,3.0)
let v2 = (3.0,2.0)

let (x,y) = (vAdd v1 v2) in printf "(%f,%f)" x y
```

(`printf` is similar to `printf` in C. It will be executed when a `.exe` file using the module is executed (or, when the module is loaded in `fsi`). It has the *side effect* of printing to stdout (typically screen). `printf` is thus *impure*)

Local Modules

The modules so far have been *top-level modules*

There can only be a single top-level module in one file

There can be several *local modules* in the same file

Local modules must have their declarations indented:

```
module Local
  let f x = ...
```

Data Type Declarations

In F# you can define your own *data types*

A first, simple example:

```
type Color = Black | Blue | Green | Cyan | Red | Magenta
           | Yellow | White
```

Here, `Color` is a *type* (just like `bool`, `int`, `int list`)

`Black`, `Blue` etc. are *constructors* (just like `true`, `17`, `[]`)

The elements of `Color` are the values `Black`, `Blue` etc.

Syntax rule: names of user-defined constructors must start with upper-case letter

We can write functions that use the `Color` values:

```
f : Color -> int

let rec f color =
    match color with
    | Black -> 17
    | Blue  -> f Black + 2
    ...
```

Pattern-matching works as usual on user-defined constructors.

(User-defined types are no different from predefined types!)

The previous example was quite limited

F# can do more than types with a small number of given elements

We can for instance define types whose elements are structured data (like tuples)

We'll do an example on next page

Example: Geometrical Shapes

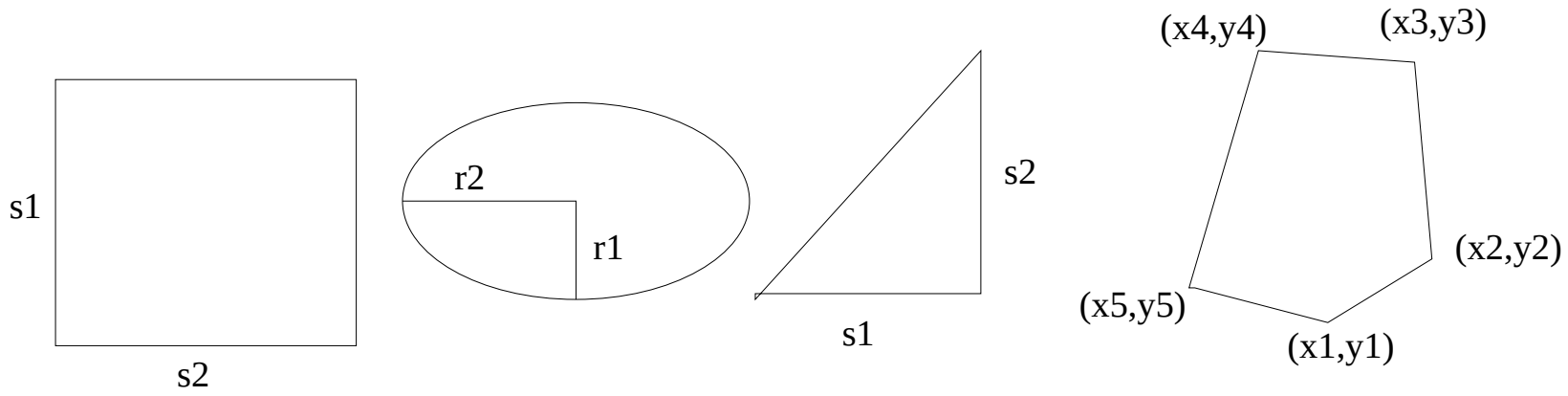
(Adapted from P. Hudak *The Haskell School of Expression: Learning Functional Programming through Multimedia*)

Say we want to represent some kinds of *geometrical shapes*

(Later, we may want to do things with them like computing their areas, or displaying them graphically, or composing them into more complex shapes)

We want to represent *rectangles*, *ellipses*, *right triangles* (90 degree angle), and general *polygons*

Rectangles, ellipses, and right triangles are characterized by two numbers, and polygons by a number of 2D-coordinates:

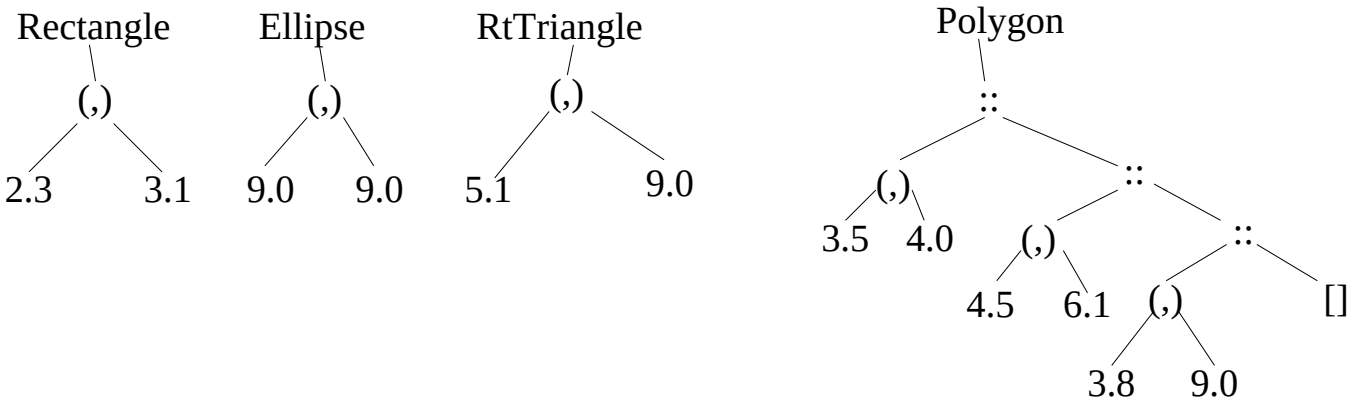


Here's the data type declaration:

```
type Shape = Rectangle of float * float
           | Ellipse of float * float
           | RtTriangle of float * float
           | Polygon of (float * float) list
```

So, for instance, `Rectangle (2.3,3.1)` represents a rectangle with sides of length 2.3 and 3.1, respectively

The constructors `Rectangle` etc. take arguments and build data structures containing these arguments



You can also think of them as unique tags:

Rectangle	2.3	3.1

So `Rectangle (2.3,3.1)` is basically the same as the tuple `(2.3,3.1)` plus a tag telling that this tuple represents a rectangle

Type Synonyms

In F#, we can declare *type synonyms*

A type synonym is a simple alias

This is useful since sometimes one uses the same data type to represent different things

With type synonyms, we can use different type names to help keep track of this.

Example:

```
type flags = bool * bool * bool
```

Type synonym declarations for our geometrical shapes:

```
type radius = float
type side   = float
type vertex = float * float
```

New definition of the `Shape` data type:

```
type Shape = Rectangle of side * side
           | Ellipse of radius * radius
           | RtTriangle of side * side
           | Polygon of vertex list
```

Functions on Shapes

Let's define a function `area : Shape -> float` that computes the area of a shape

Solution on the next few slides . . .

`area` can be defined case by case by pattern-matching on different constructors

```
area shape = match shape with
             ....
```

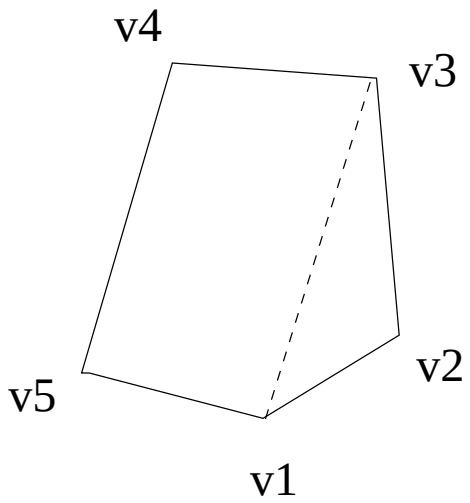
Easy cases first:

```
| Rectangle (s1,s2)  -> s1*s2
| RtTriangle (s1,s2) -> s1*s2/2.0
| Ellipse (r1,r2)    -> pi*r1*r2
```

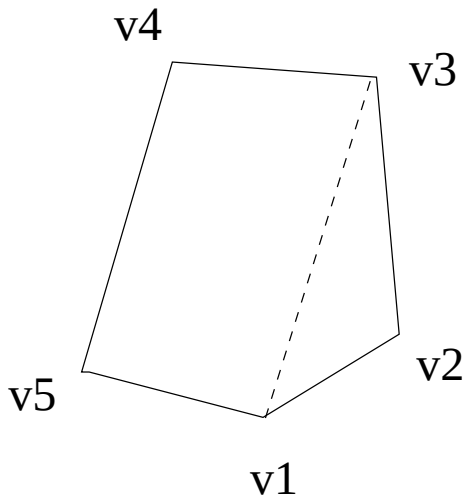
(Assuming `pi` is defined in the module we're working in)

What about polygons?

Three corners or more: compute it by cutting a triangle, computing its area, and adding to area of rest of polygon (which is also a convex polygon)



Recursive function: how do we know that it will terminate?



1. We start with a finite number of corners
2. One corner removed for each cut
3. Thus, sooner or later there are only three corners left
4. That is a single triangle, we then compute the area of that triangle and return it

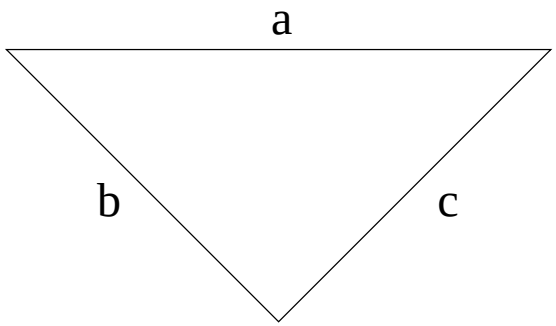
Solution:

Assume for now a function `triArea` that compute the area of a triangle given its corners

```
| Polygon (v1::v2::v3::vs)
  -> (triArea v1 v2 v3) + area (Polygon (v1::v3::vs))
| (Polygon _) -> 0.0
```

(The first case takes care of the case where the polygon has at least three corners. The last case takes care of the case when it has two or less corners)

`triArea` is computed with *Heron's formula*:



$$A = \sqrt{s(s-a)(s-b)(s-c)}, \text{ where } s = \frac{1}{2}(a+b+c)$$

(This is classical geometry. Heron lived 2000 years ago.)

We have the vertices but not the length of the sides between them

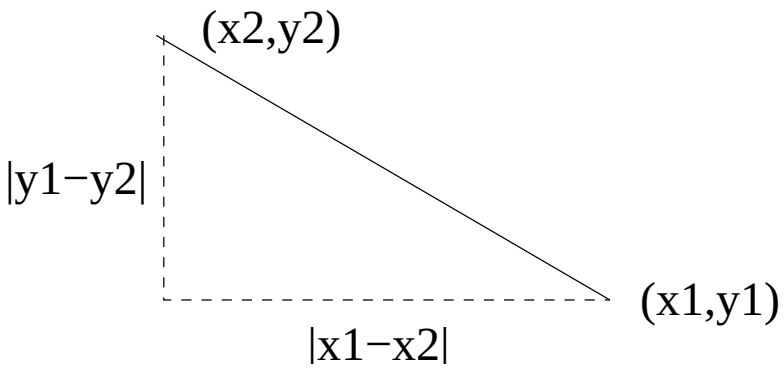
Assume for now a function `distBetween` that computes the distance between two vertices:

```
let triArea v1 v2 v3 = let a = distBetween v1 v2
                       let b = distBetween v2 v3
                       let c = distBetween v3 v1
                       let s = 0.5*(a+b+c)
                       in sqrt (s*(s-a)*(s-b)*(s-c))
```

(Note how we can make multiple local definitions using `let`. With the default simplified syntax, we can even drop "`in`")

Finally,

```
let distBetween (x1,y1) (x2,y2) = sqrt ((x1-x2)**2.0 + (y1-y2)**2.0)
```



A Note on Programming Style

In the polygon case, we used smaller functions (`triArea`, `distBetween`) to compute results needed to compute the whole area

This is a style of programming supported well by functional programming languages like F#: define (or use predefined) small, general functions to successively compose the desired solution

Record Types

F# also has *records* (similar to structs in C, or simple objects)

Basically, a record is a tuple where every field has a name

A declaration of a record type for representing vertices in polygons (2D-coordinates):

```
type Vertex = { x : float; y : float }
```

Access is by “dot” notation, like:

```
let vlen coord = sqrt (coord.x**2.0 + coord.y**2.0)
vlen : Vertex -> float
```

Record fields can *not* be accessed by pattern matching

Creating Records

A record is created by giving the value for each field:

```
{ x = 3.0; y = 4.0 }
```

Order does not matter:

```
{ y = 4.0; x = 3.0 }
```

A function that converts a pair of floats into a `Vertex`:

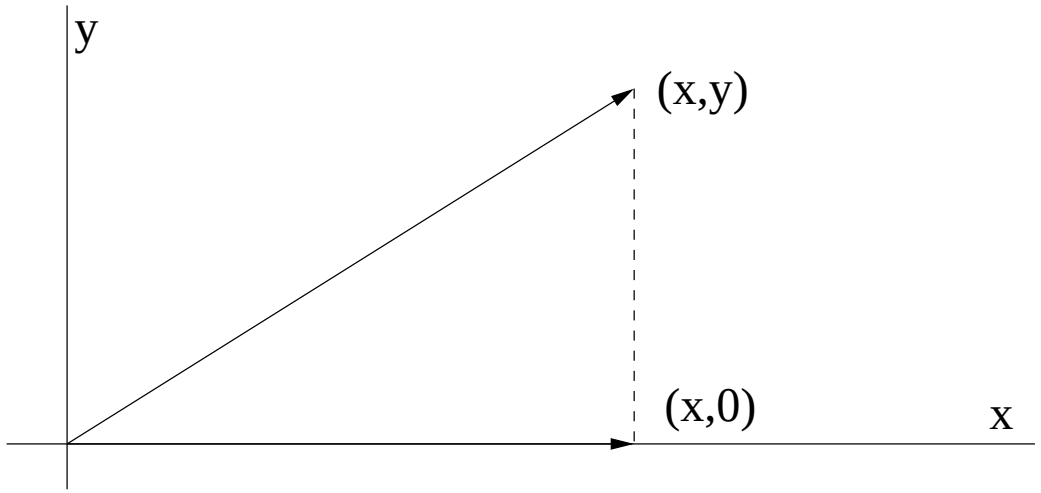
```
let pair2Vertex (a,b) = { x = a; y = b }
pair2Vertex : (float * float) -> Vertex
```

Creating Records

A construct to create a new record from an old one, by replacing the values of some fields:

```
let project v = { v with y = 0.0 }
```

```
project { x = 3.0; y = 4.0 } ⟹ { x = 3.0; y = 0.0 }
```



Exercise

An exercise:

Redefine the `Shape` data type to use records rather than tuples

Then redefine the `area` function to use this new data type instead!

Suitable to do at home . . .