OCaml Datatypes

COS 326

Speaker: Andrew Appel

Princeton University

OCaml So Far

- We have seen a number of basic types:
  - int
  - float
  - char
  - string
  - bool
- We have seen a few structured types:
  - pairs
  - tuples
  - options
  - lists
- In this lecture, we will see some more general ways to define our own new types and data structures

Type Abbreviations

- We have already seen some type abbreviations:

```
type point = float * float
```

- These abbreviations can be helpful documentation:

```
let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

- But they add nothing of *substance* to the language
  - they are equal in every way to an existing type

Type Abbreviations

- We have already seen some type abbreviations:

```
type point = float * float
```

- As far as OCaml is concerned, you could have written:

```
let distance (p1:float*float)
             (p2:float*float) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

- Since the types are equal, you can *substitute* the definition for the name wherever you want
  - we have not added any new data structures

DATA TYPES

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```

a value with type my_bool is one of two things:

- Tru, or
- Fal

read the "|" as "or"

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```

Tru and Fal are called "constructors"

a value with type my_bool is one of two things:

- Tru, or
- Fal

read the "|" as "or"

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```

```
type color = Blue | Yellow | Green | Red
```

there's no need to stop at 2 cases; define as many alternatives as you want

Data types

- OCaml provides a general mechanism called a data type for defining new data structures that consist of many alternatives

```
type my_bool = Tru | Fal
```

```
type color = Blue | Yellow | Green | Red
```

- Creating values:

```
let b1 : my_bool = Tru
let b2 : my_bool = Fal
let c1 : color = Yellow
let c2 : color = Red
```

use constructors to create values

Data types

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```

- Using data type values:

```
let print_color (c:color) : unit =
  match c with
  | Blue ->
  | Yellow ->
  | Green ->
  | Red ->
```

use pattern matching to determine which color you have; act accordingly

Data types

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```

- Using data type values:

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Green -> print_string "green"
  | Red -> print_string "red"
```

Data types

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```

- Using data type values:

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Green -> print_string "green"
  | Red -> print_string "red"
```

Why not just use strings to represent colors instead of defining a new type?

Data types

```
type color = Blue | Yellow | Green | Red
```

oops!:

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Red -> print_string "red"
```

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Green

Data types

```
type color = Blue | Yellow | Green | Red
```

oops!:

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Red -> print_string "red"
```

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Green

OCaml's datatype mechanism allows you to create types that contain *precisely* the values you want!

Data types

```
type color = Blue | Yellow | Green | Red
```

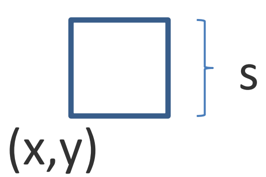
This is like an “enumeration” type in Pascal, C, Java, ...

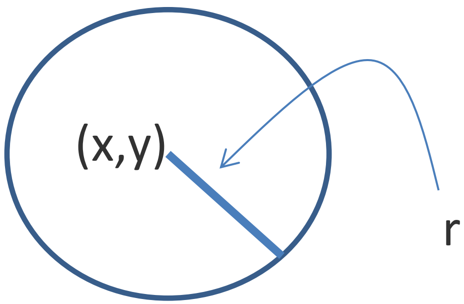
Data Types Can Carry Additional Values

- Data types are more than just enumerations of constants:

```
type point = float * float

type simple_shape =
   Circle of point * float
 | Square of point * float
```

- Read as: a simple_shape is either:
  – a Circle, which contains a pair of a point and float, or
  – a Square, which contains a pair of a point and float





Data Types Can Carry Additional Values

- Data types are more than just enumerations of constants:

```
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let origin : point = (0.0, 0.0)

let circ1  : simple_shape = Circle (origin, 1.0)
let circ2  : simple_shape = Circle ((1.0, 1.0), 5.0)
let square : simple_shape = Square (origin, 2.3)
```

Data Types Can Carry Additional Values

- Data types are more than just enumerations of constants:

```
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let simple_area (s:simple_shape) : float =
  match s with
  | Circle (_, radius) -> 3.14 *. radius *. radius
  | Square (_, side) -> side *. side
```

Compare

- Data types are more than just enumerations of constants:

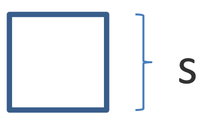
```
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let simple_area (s:simple_shape) : float =
  match s with
  | Circle (_, radius) -> 3.14 *. radius *. radius
  | Square (_, side) -> side *. side
```

```
type my_shape = point * float

let simple_area (s:my_shape) : float =
  (3.14 *. radius *. radius)  ?? or ??  (side *. side)
```

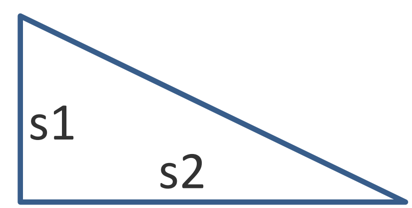
More General Shapes

```
type point = float * float

type shape =
    Square of float
  | Ellipse of float * float
  | RtTriangle of float * float
  | Polygon of point list
```

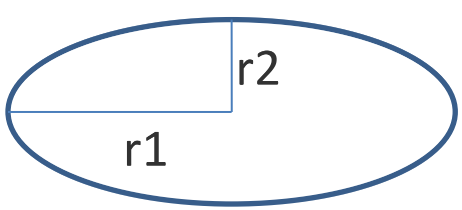
Square s =



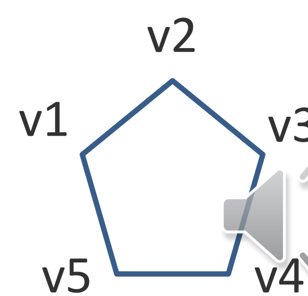
RtTriangle (s1, s2) =



Ellipse (r1, r2) =



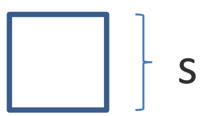
Polygon [v1; ...;v5] =

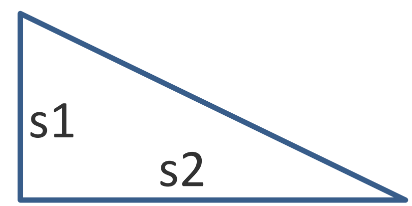


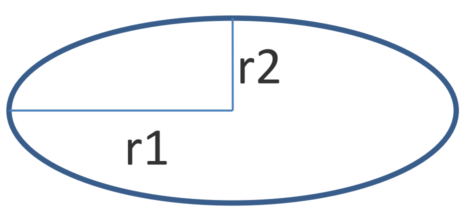
More General Shapes

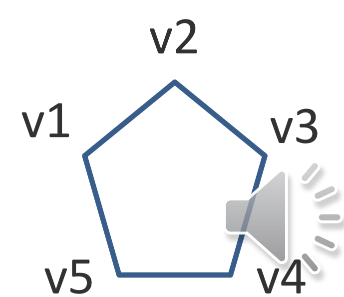
```
type point = float * float
type radius = float
type side = float

type shape =
    Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list
```

Type abbreviations can aid readability

Square s = 

RtTriangle (s1, s2) = 

Ellipse (r1, r2) = 

RtTriangle [v1; ...;v5] = 

More General Shapes

```
type point = float * float
type radius = float
type side = float

type shape =
    Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list
```

Square builds a shape from a single side

RtTriangle builds a shape from a pair of sides

```
let sq   : shape = Square 17.0
let ell  : shape = Ellipse (1.0, 2.0)
let rt   : shape = RtTriangle (1.0, 1.0)
let poly : shape = Polygon [(0., 0.); (1., 0.); (0.; 1.)]
```

they are all shapes; they are constructed in different ways

Polygon builds a shape from a list of points (where each point is itself a pair)

More General Shapes

```
type point = float * float
type radius = float
type side = float

type shape =
    Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list
```

a data type also defines a pattern for matching

```
let area (s : shape) : float =
  match s with
   | Square s ->
   | Ellipse (r1, r2)->
   | RtTriangle (s1, s2) ->
   | Polygon ps ->
```

Square carries a value with type float so s is a pattern for float values

RtTriangle carries a value with type float * float so (s1, s2) is a pattern for that type

More General Shapes

```
type point = float * float
type radius = float
type side = float

type shape =
    Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list
```

a data type also defines a pattern for matching

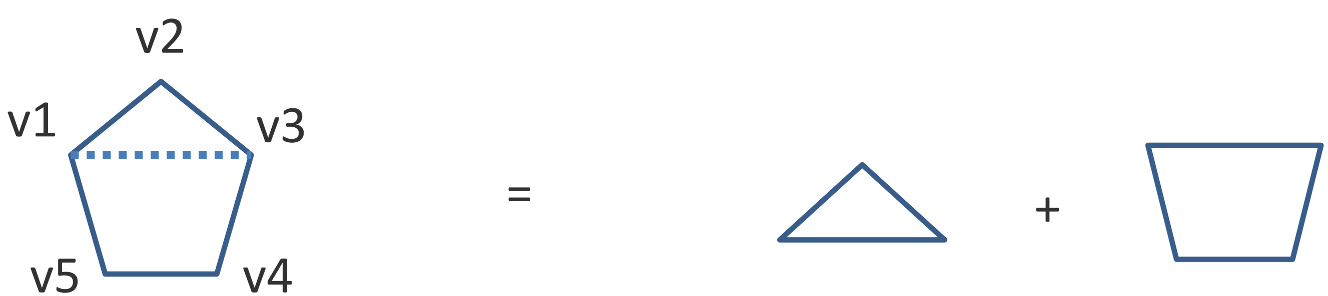
```
let area (s : shape) : float =
  match s with
  | Square s -> s *. s
  | Ellipse (r1, r2)-> pi *. r1 *. r2
  | RtTriangle (s1, s2) -> s1 *. s2 /. 2.
  | Polygon ps -> ???
```

Computing Area

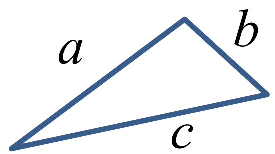
- How do we compute polygon area?
- For convex polygons:
  - Case: the polygon has fewer than 3 points:
    - it has 0 area! (it is a line or a point or nothing at all)
  - Case: the polygon has 3 or more points:
    - Compute the area of the triangle formed by the first 3 vertices
    - Delete the second vertex to form a new polygon
    - Sum the area of the triangle and the new polygon

Computing Area

- How do we compute polygon area?
- For convex polygons:
  - Case: the polygon has fewer than 3 points:
    - it has 0 area! (it is a line or a point or nothing at all)
  - Case: the polygon has 3 or more points:
    - Compute the area of the triangle formed by the first 3 vertices
    - Delete the second vertex to form a new polygon
    - Sum the area of the triangle and the new polygon
- Note: This is a beautiful inductive algorithm:
  - the area of a polygon with n points is computed in terms of a smaller polygon with only n-1 points!



How do you compute the area of a triangle?



$$\text{Let} \quad s = \frac{a+b+c}{2}$$

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

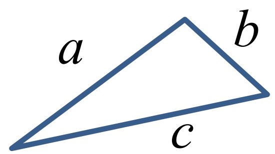
Heron’s formula

Published by Heron of Alexandria (Egypt), 60 A.D.

Probably known to Archimedes (Syracuse, Sicily), 220 B.C.

Published independently by Qin Jiushao 秦九韶, (Sichuan province) 1247

How do you compute the area of a triangle?



$$\text{Let } s = \frac{a+b+c}{2}$$

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

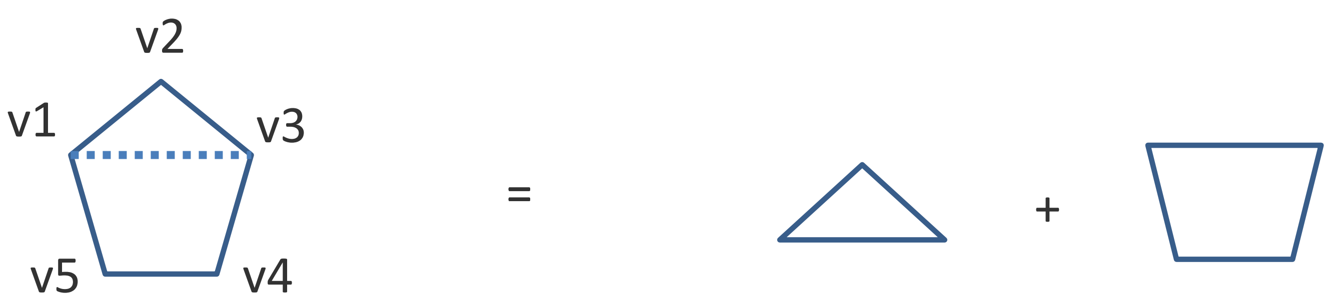
```
let tri_area (p1:point) (p2:point) (p3:point) : float =
  let a = distance p1 p2 in
  let b = distance p2 p3 in
  let c = distance p3 p1 in
  let s = 0.5 *. (a +. b +. c) in
  sqrt (s *. (s -. a) *. (s -. b) *. (s -. c))
```

Computing Area

```
let poly_area (ps : point list) : float =
  match ps with
  | p1 :: p2 :: p3 :: tail ->
     tri_area p1 p2 p3 +. poly_area (p1::p3::tail)
  | _ -> 0.
```

This pattern says the list has at least 3 items

```
let area (s : shape) : float =
  match s with
  | Square s -> s *. s
  | Ellipse (r1, r2)-> r1 *. r2
  | RtTriangle (s1, s2) -> s1 *. s2 /. 2.
  | Polygon ps -> poly_area ps
```



Summary

A *datatype* t has *constructors* c_1 c_2 c_3 ...

Each constructor may carry a value (like Square(s))
or be a *constant constructor* (like Green)

We *build* values of type t by applying constructors to values
(or by applying constant constructors to nothing)

We *examine* values of type t by *pattern-matching.*