Type synonyms

Syntax: `type id = t`

- Anywhere you write **`t`**, you can also write **`id`**
- The two names are *synonymous*

e.g.

```
type point  = float * float
type vector = float list
type matrix = float list list
```

Type synonyms

```
type point = float*float

let getx : point -> float =
  fun (x,_) -> x

let pt : point = (1.,2.)
let floatpair : float*float = (1.,3.)

let one  = getx pt
let one' = getx floatpair
```

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 O'Caml 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

- O'Caml 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

- O'Caml 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

- O'Caml 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

- O'Caml 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"
```

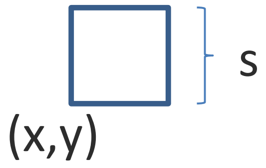
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Green

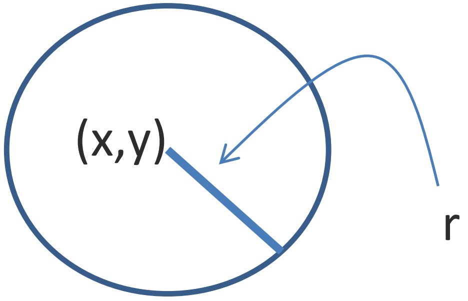
Data Types Can Carry Additional Values

- 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

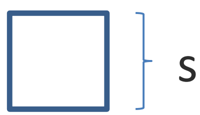
- Data types are more than just enumerations of constants:

```
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
```

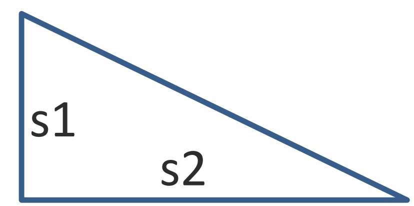
More General Shapes

```
type point = float * float

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

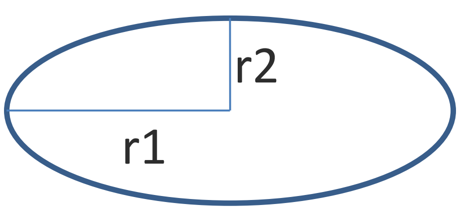
Square s =



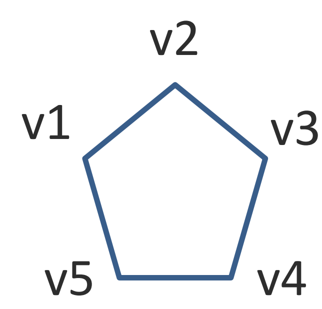
RtTriangle (s1, s2) =



Ellipse (r1, r2) =



Polygon [v1; ...;v5] =

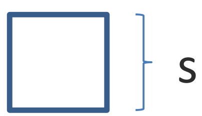


More General Shapes

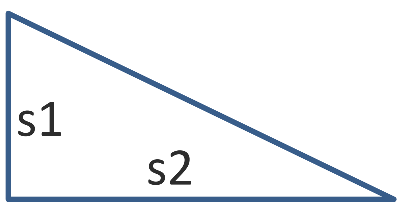
```
type point = float * float
type radius = float
type side = float

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

Type abbreviations can aid readability

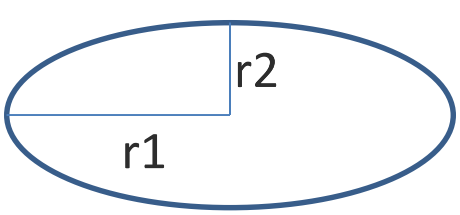
Square s =



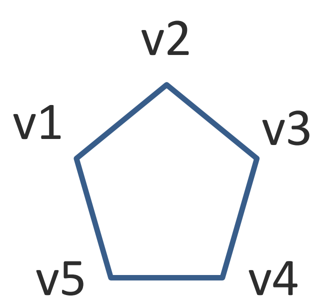
RtTriangle (s1, s2) =



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

```
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.)]
```

RtTriangle builds a shape from a pair of sides

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 ->
```

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

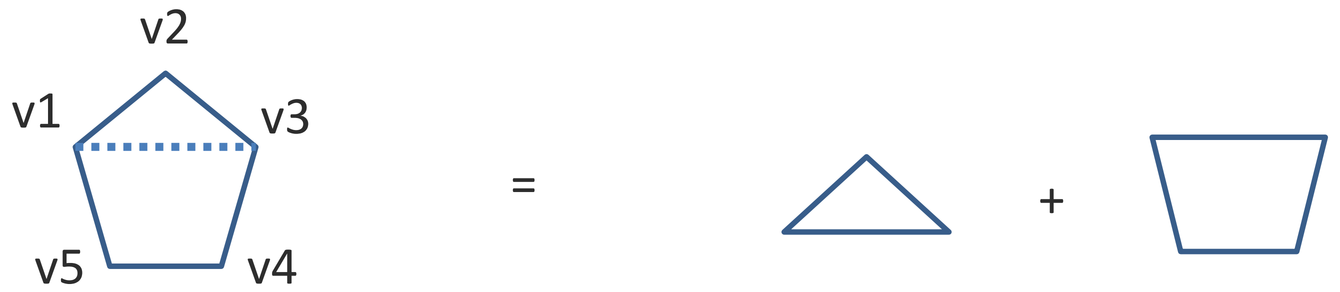
```
type point = float * float
type radius = float
type side = float

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

```
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 -> ???
```

a data type also defines a pattern for matching

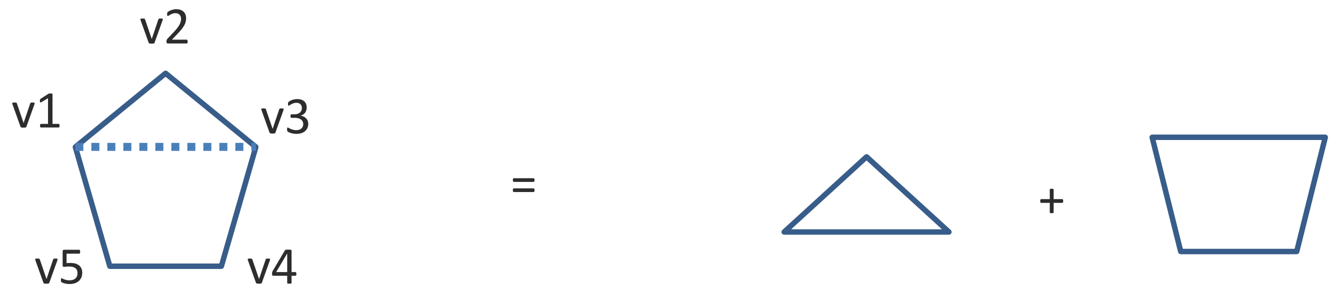
Computing Area

- 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!

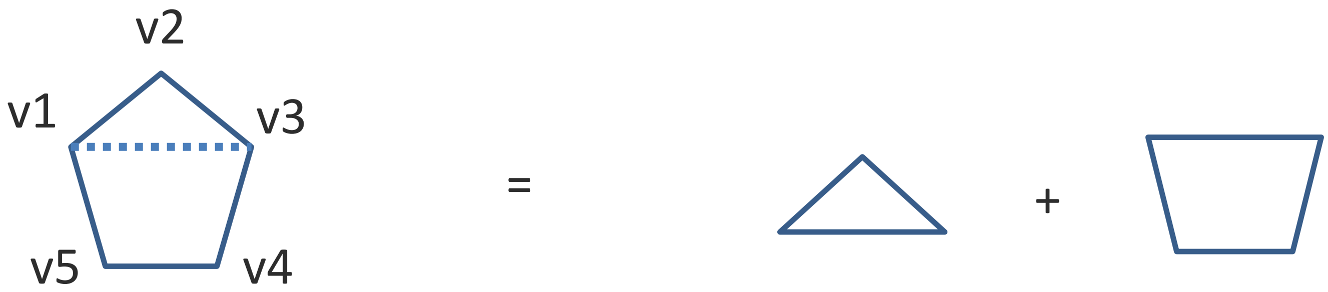


Computing Area

```
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
```

This pattern says the list has at least 3 items

```
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.
```



Computing Area

```
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))
```

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

```
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
```

INDUCTIVE DATA TYPES

Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree

Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree

```
type key = string
type value = int

type tree =
  Leaf
| Node of key * value * tree * tree
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf ->
  | Node (k', v', left, right) ->
```

Again, the type definition specifies the cases you must consider

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Inductive data types

```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
      if k < k' then
        Node (k', v', insert left k v, right)
      else if k > k' then
        Node (k', v', left, insert right k v)
      else
        Node (k, v, left, right)
```

Implement lists with variants

```
type intlist = Nil | Cons of int * intlist

let emp = Nil
let l3 = Cons (3, Nil)  (* 3::[] or [3]*)
let l123 = Cons(1, Cons(2, l3)) (* [1;2;3] *)

let rec sum (l:intlist) =
  match l with
   | Nil -> 0
   | Cons(h,t) -> h + sum t
```

Implement lists with variants

```
let rec length = function
   | Nil -> 0
   | Cons (_,t) -> 1 + length t
(* length : intlist -> int *)

let empty = function
   | Nil -> true
   | Cons _ -> false
(* empty: intlist -> bool *)
```

Implement lists with variants

```
let rec fold_right f l acc =
  match l with
   | Nil -> acc
   | Cons(h,t) -> f h (fold_right f t acc)
(* fold_right:
   (int -> 'a -> 'a)
   -> intlist -> 'a -> 'a *)

let sumr l = fold_right (+) l 0
(* empty: intlist -> int *)
```

Implement lists with variants

```
let hd = function
  | Nil -> ???
  | Cons(h,t) -> h
```

One possibility is to return an option:

```
let hd = function
  | Nil -> None
  | Cons(h,t) -> Some h
(* hd: intlist -> int option *)
```

But the standard library throws an exception...

EXCEPTIONS

Example: implement hd

```
let hd = function
  | Nil -> raise (Failure "empty")
  | Cons(h,t) -> h

# hd Nil;;
Exception: (Failure empty).

let head_or_zero lst =
  try hd lst with
  | Failure s -> 0

# head_or_zero Nil;;
- : int = 0
```

Exceptions: Syntax

Definition:

```
exception E
exception E of t
```

Raise (aka throw):

```
raise e
```

Catch (aka handle):

```
try e with
| p1 -> e1
| ...
| pn -> en
```

Exception: Type checking

New kind of type: `exn`

if `E` is defined as `exception E` then `E : exn`

if `E` is defined as `exception E of t` and `e : t`

then `E e : exn`

Raise:

if `e:exn` then `raise e` may have any type `t`

Catch:

if `e` and `p1..pn` and `e1..en` all have type `t`

then `try e with p1 -> e1 | ... | pn -> en`

has type `t`

Exceptions: Evaluation

Raise:

If `e ==> v` then `raise e` produces an *exception packet* containing `v` that propagates upward through the call stack to a handler.

Catch:

`try e with p1 -> e1 | ... | pn -> en`

If `e ==> v` then the `try` expression evaluates to `v`.

If evaluation of `e` produces an exception packet, behave like a pattern match on the value in that packet.

But if none of the patterns matches, re-raise the exception, thus propagating it upwards.

Exceptions in standard library

```
exception Invalid_argument of string
```

raised by library functions to signal that the given arguments do not make sense

```
exception Failure of string
```

raised by library functions to signal that they are undefined on the given arguments

Convenience function in library:

```
let failwith : string -> 'a =
  fun s -> raise (Failure s)
```

Inductive data types: Another Example

- Recall, we used the type "int" to represent natural numbers
  - but that was kind of broken: it also contained negative numbers
  - we had to use a dynamic test to guard entry to a function:

```
let double (n : int) : int =
  if n < 0 then
    raise (Failure "negative input!")
  else
    double_nat n
```

  - it would be nice if there was a way to define the natural numbers exactly, and use OCaml's type system to guarantee no client ever attempts to double a negative number

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

```
type nat = Zero | Next of nat
```

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

```
type nat = Zero | Succ of nat

let rec nat_to_int (n : nat) : int =
 match n with
   Zero -> 0
 | Succ n -> 1 + nat_to_int n
```

Inductive data types

- Recall, a natural number n is either:
  - zero, or
  - m + 1
- We use a data type to represent this definition exactly:

```
type nat = Zero | Next of nat

let rec nat_to_int (n : nat) : int =
 match n with
   Zero -> 0
 | Next n -> 1 + nat_to_int n

let rec double_nat (n : nat) : nat =
  match n with
   | Zero -> Zero
   | Succ m -> Succ (Succ(double_nat m))
```

A Note on
Parameterized Type Definitions

```
type ('key, 'val) tree =
    Leaf
  | Node of 'key * 'val * ('key, 'val) tree * ('key, 'val) tree


type 'a stree = (string, 'a) tree


type sitree = int stree
```

<u>General form:</u>

definition:

```
type 'x f = body
```

use:

```
arg f
```

<u>A Better Notation:</u>

definition:

```
type f x = body
```

use:

```
f arg
```

Take-home Message

- Think of parameterized types like functions:
  - a function that take a type as an argument
  - produces a type as a result

- Theoretical basis:
  - System F-omega
  - a typed lambda calculus with general type-level functions as well as value-level functions

Summary

- OCaml datatypes: a powerful mechanism for defining complex data structures:
  - They are precise
    - contain exactly the elements you want, not more elements
  - They are general
    - recursive, non-recursive (mutually recursive and polymorphic)
  - The type checker helps you detect errors
    - missing cases in your functions
  - Next time: help in program evolution