

Lecture 3 — User-defined types

- User-defined types in OCaml
- Defining trees

Type abbreviations in OCaml

- OCaml allows new names to be introduced as abbreviations for types:

$\text{type } t = te$

- te is a type expression:

$te = \text{int} \mid \text{float} \mid \dots \mid te * te * \dots * te$
 $\mid te \text{ list} \mid te \rightarrow te$

Defining new types

- OCaml allows you to create new types by defining a set of constructors:

$$\text{type } t = C_1 \text{ [of } te_1] \mid \dots \mid C_n \text{ [of } te_n]$$

where $C_1, \dots, C_n$ are *constructors* (identifiers starting with capital letters).

- The above declaration creates a new type, called t .
- It also creates new functions that construct values of type t :
 - $C_1 : te_1 \rightarrow t$
 - $\dots$
 - $C_n : te_n \rightarrow t$

Defining new types (cont.)

- Think of values of type t as tuples combined with a *tag* — a number between 1 and n — saying which kind of t -typed value it is.
- Functions on values of type t can be defined using pattern-matching:

```
let f x = match x with
    C1( $x, \dots, y$ ) ->  $e_1$ 
  | C2( $x, \dots, y$ ) ->  $e_2$ 
  | ...
  | C $n$ ( $x, \dots, y$ ) ->  $e_n$ 
```

Type definition example

```
type shape = Circle of float  
           | Square of float  
           | Triangle of float * float * float
```

```
let c = Circle 5.7  
let t = Triangle (2.0, 3.0, 4.0)
```

```
let shape_to_string S =  
  match s with  
  | Circle r -> "circle" ^ string_of_float r  
  | Square t -> "square" ^ string_of_float t  
  | Triangle (s1, s2, s3) ->  
    "triangle(" ^ float_to_string s1 ^ "," ^  
    string_of_float s2 ^ "," ^  
    string_of_float s3 ^ ")"
```

Type definition exercise

- Given this new type

```
type PersonalInfo = Address of int * string  
                  | Phone of string | Age of int
```

define function `street: PersonalInfo → string` **that returns the street name for an address, and the empty string for any other kind of value:**

Recursive type definitions

- In this type definition:

$$\text{type } t = C_1 \text{ [of } te_1] \mid \dots \mid C_n \text{ [of } te_n]$$

the type expressions te_i can contain t , making the type declaration recursive. This allows for the definition of infinite data types, such as lists and trees, e.g.

```
type mylist = Empty | Cons of int * mylist  
let list1 = Cons (3, Cons (4, Empty))
```

- Ex: write the function $\text{sum} : \text{mylist} \rightarrow \text{int}$.

Defining trees

- **Binary trees (with integer labels):**

```
type bintree = Empty
              | Node of int * bintree * bintree
```

```
let tree1 = Node (3,
                  Node (6, Empty, Empty),
                  Node (7, Empty, Empty));;
```

- **Arbitrary trees (with integer labels):**

```
type tree = Node of int * tree list
```

```
let smalltree = Node (3, [])
let bigtree = Node (3, [Node(...), Node(...), ...])
```

Exercises: Functions on binary trees

```
type bintree = Empty  
              | Node of int * bintree * bintree
```

- Define **isLeaf**: $\text{bintree} \rightarrow \text{bool}$

- Define **sum**: $\text{bintree} \rightarrow \text{int}$

Polymorphic types

- We can define a type of binary trees with labels of any type (but all the same type for any particular tree):

```
type 'a bintree = Empty  
                | Node of 'a * 'a bintree * 'a bintree
```

```
let x = Node("ben", Empty, Empty)  
let y = Node(4.5, Empty, Empty)
```

- The `sum` function defined above still works, when applied to a value of type `int bintree`.
- bintrees are homogeneous, e.g.

```
Node("ben", Node(4, Empty, Empty), Empty)
```

gives a type error.

Mutually recursive types

- Sometimes two user-defined types are mutually interdependent: values of either type can contain values of the other type. To define mutually-recursive types, give both type declarations separated by the word `and`:

```
type t = C1 of te1 | ...  
and u = D1 of te1' | ...
```

Abstract syntax

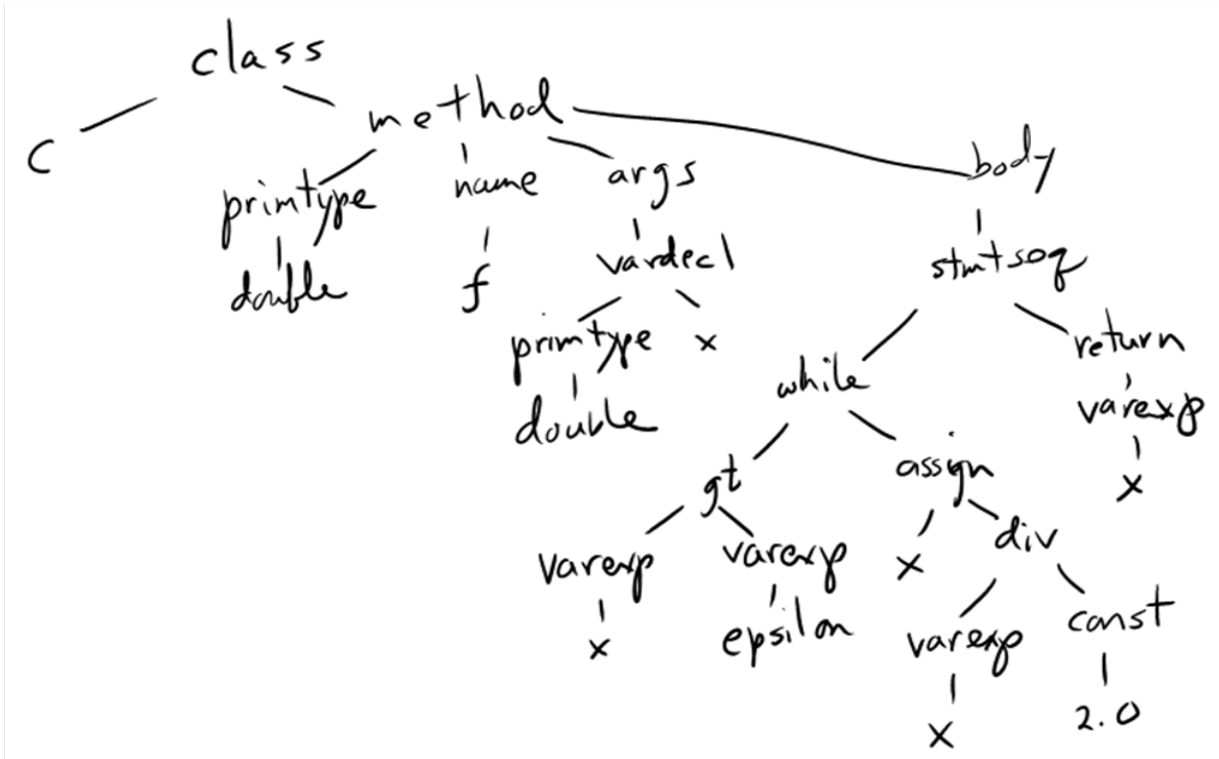
- *Abstract syntax* is a tree representation of the syntactic structure of programs.
- The internal nodes of abstract syntax trees are labelled with names, called *abstract syntax operators*; the leaf nodes are labelled with strings, ints, etc.
- The specific trees used to represent programs in a given language are determined by the person writing the language processor (e.g. compiler).
- For example, this program in Java:

```
class C {  
    double f (double x) {  
        while (x > epsilon) x = x/2.0;  
        return x; } }  

```

Abstract syntax (cont.)

would be represented by a tree something like this:



In OCaml, type definitions can be used to define abstract syntax, and pattern-matching can be used to define functions on abstract syntax trees.