

CS 42I Lecture 3

- ▶ Today's class: Types in OCaml and abstract syntax
 - ▶ Type declaration in OCaml
 - ▶ Trees
 - ▶ Polymorphic types
 - ▶ Abstract syntax

Type declaration in OCaml

- ▶ First, type expressions are:

- ▶ $te = int \mid string \mid unit \mid \dots \mid te \text{ list} \mid te * te * \dots * te$

Type declaration in OCaml

- ▶ `type t = te`
 - ▶ After this, `t` is an abbreviation for `te`
- ▶ `type t = C1 [of te1] | ... | Cn [of ten]`
where `C1`, ..., `Cn` are *constructor names* – names that start with a capital letter

Example – enumerated types

▶ Ex.

```
type weekday = Mon | Tues | Wed | Thurs | Fri;;  
let today = Tues;;  
let weekday_to_string d =  
  match d with  
    | Mon -> "Monday"  
    | Tues -> "Tuesday"  
    | ... ;;
```

Corresponds to “enum” type in C, C++:

```
typedef enum {Mon, Tues, Wed, Thurs, Fri} weekday;
```

Example – disjoint unions

▶ Ex.

```
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)
```

- ▶ This corresponds to what is called *discriminated union*, *tagged union*, *disjoint union*, or *variant record*.

Example – disjoint unions (cont.)

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

How to do this in C

```
struct shape {
    int type_of_shape;
    union {
        struct {float radius;}
        struct {float side;}
        struct {float side1, side2, side3;} triangle;
    } shape_data;
}
```

Shape_to_string function would look like this:

```
switch (type_of_shape){
    case 0: cout << "circle" << s.shape_data.radius;
    ... etc. ...
```

How to do this in Java – method 1

```
class Shape{  
    float x; // radius or side  
    float side2, side3;  
    int shape_type;  
    Shape(int i, float f){  
        shape_type = i;  
        x = f; }  
    Shape(float, float, float){  
        shape_type = 2; x = ...;  
        side2 = ...; side3 = ...;  
    }  
}
```

- ▶ `shape_to_string` looks the same as in C.

How to do this in Java – method 2

```
class Shape{
    virtual string shape_to_string();
}
class Circle extends Shape {
    float radius;
    Circle(float r) {radius = r;}
    string shape_to_string(){
        return "circle" + radius; }
}
class Square extends Shape {
    float side;
    Square (float s) {side = s;}
    string shape_to_string(){
        return "square" + side; }
}
```

Recursive type definitions in OCaml

In type $t = C \text{ of } e \mid \dots$, e can include t .

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

```
let rec sum x = match x with
  Empty -> 0
  | Cons(y,ys) -> y + sum ys
```

Defining trees

Binary trees (with integer labels):

```
type bintree = Empty
              | BTreeNode of int * bintree * bintree
let tree1 = BTreeNode (3,
                      BTreeNode (6, Empty, Empty), . . .);;
```

Arbitrary trees (with integer labels):

```
type tree = Node of int * tree list
let smalltree = Node (3, [])
let bigtree = Node (3, [Node(...), Node(...), ...])
```

Trees

Ex. Create a list of all the integers in a tree

Use homework function `flatten : (int list) list -> int list`

```
let rec flatten_tree (Node (n, kids)) =  
  let rec flatten_list tlist = match tlist with  
    [] -> []  
  | (t :: ts) -> flatten_tree t :: flatten_list ts  
  in n :: flatten (flatten_list kids)
```

Defining polymorphic types

```
type 'a bintree = Empty
                | Node of 'a * 'a bintree * 'a bintree
let x = Node("ben", Empty, Empty)
let y = Node(4.5, Empty, Empty)
```

▶ Mutually-recursive types

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

Abstract syntax

- ▶ “Deep” structure of program – represents nesting of fragments within other fragments in the “cleanest” way possible. Can define as a type in Ocaml, e.g.

```
type stmt = Assign of string * expr
          | If of expr * stmt * stmt
and expr = Int of int | Var of string
          | Plus of expr*expr | Greater of expr*expr
```

“if (x>0) y=y+1; else z=x;” →

```
If(Greater(Var "x", Int 0),
    Assign("y", Plus(Var "y", Int 1)),
    Assign("z", Var "x"))
```

-
- ▶ Lecture 3