CS 421 Lecture 3

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

Type declaration in OCaml

OCaml allows news names to be introduced as abbreviations for types:

type t = te

- te is a *type expression:*
- te = int | string | unit | … | te list | te * te * … * te

Type declaration in OCaml

- More importantly, it allows you to create new types by defining a set of *constructors*:
- type t = C_1 [of te_1] | … | C_n [of te_n]
  where C_1 , … , C_n are the constructors, (Constructor names must start with a capital letter.)
- Values of type t are created by applying C_1 to value of type te_1, or C_2 to value of type te_2, etc.

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)
```

- (Note: Triangle 2.0 3.0 4.0 is type error.)
- 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{
  abstract 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; }
}
```

```
Shape sh;
if (…)
    sh = new Circle(…);
else
    sh = new Square(…);
…
sh.shape_to_string()
```

Recursive type definitions in OCaml

In type t = C of te | … , te can include t.

```
type mylist = Empty | Cons of int * mylist
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
             | BTNode of int * bintree * bintree
let tree1 = BTNode (3,
               BTNode (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 tlis = match tlis with
        [] -> []
      | (t :: ts) -> flatten_tree t :: flatten_list ts
    in n :: flatten (flatten_list kids)
```

Syntactic note: `flatten_tree Node(…,…)` would be interpreted as `(flatten_tree Node)(…,…).` Since Node has type (int * tree list) -> int list, and the argument to flatten_tree should be tree, this is a type error. Need to write `flatten_tree (Node(…, …))`

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)
```

- Although bintree is polymorphic, can still define functions that apply only to some bintrees (as you can for lists), e.g.

```
let rec sum t = match t with
    Empty -> 0  | Node(i,t1,t2) -> i + sum t1 + sum t2
sum: int bintree -> int
```

Mutually-recursive types

- Mutually-recursive types

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

- Example given below

Abstract syntax

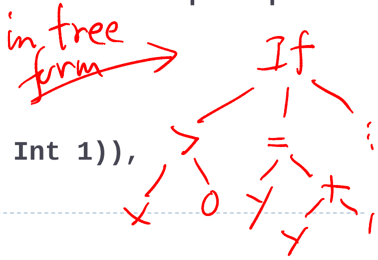
- “Deep” structure of program – represents nesting of syntactic units within other syntactic units as a tree. Can define as a type in Ocaml, e.g.

```
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”))
```

in tree form



Abstract syntax (cont.)

- Example: Function to find all the variables used in an abstract syntax tree (AST):

```
let rec vars s = match s with
     Assign(x,e) -> x :: evars e
   | If(e,s1,s2) -> evars e @ vars s1 @ vars s2
and evars e = match e with
     Int i -> []
   | Var x -> [x]
   | Plus(e1,e2) -> evars e1 @ evars e2
   | Greater(e1,e2) -> evars e1 @ evars e2
```

Abstract syntax (cont.)

- Abstract syntax for a part of Ocaml gives example of mutually-recursive type definitions:

```
type decl = Decl of (string * expr) list
and expr = Int of int | Var of string
         | Plus of expr * expr
         | Let of decl * expr
```

E.g. `“let x = 3 and y = 5 in x+y”` would have abstract syntax tree:

```
        Let(Decl[(“x”, Int 3), (“y”, Int 5)],
            Plus(Var “x”, Var “y”)
```

MP 2 – Abstract syntax for MiniJava

(N.B. Do not use this definition as a reference for MP2, as the definition used there may differ slightly from this.)

type program = Program of (class_decl list)

and class_decl = Class of id * id * (var_decl list) * (method_decl list)

and method_decl = Method of exp_type * id * ((exp_type * id) list) * (var_decl list) * (statement list) * exp

and var_decl = Var of exp_type * id

MP 2 – Abstract syntax for MiniJava

```
and statement = Block of (statement list)
    | If of exp * statement * statement
    | While of exp * statement
    | Println of exp
    | Assignment of id * exp
    | ArrayAssignment of id * exp * exp
    | Break
    | Continue
    | Switch of exp * ((int * (statement list)) list)
```

MP 2 – Abstract syntax for MiniJava

```
and exp = Operation of exp * binary_operation * exp
    | Array of exp * exp | Length of exp
    | MethodCall of exp * id * (exp list)
    | Id of id  | This | NewArray of exp_type * exp
    | NewId of id | Not of exp | Null | True | False
    | Integer of int | String of string | Float of float
and binary_operation = And | Or | LessThan
    | Plus | Minus | Multiplication | Division
and exp_type = ArrayType of exp_type | BoolType
    | IntType | ObjectType of id | StringType | FloatType
and id = string
```

MP 2 – Abstract syntax for MiniJava

Functions defined on this abstract syntax will generally consist of several mutually recursive functions:

```
let rec f_program (Program cds) = …
and f_classdecls cds = match cds with [] -> … | (c::cs) -> …
and f_classdecl (Class(name, superclass, fields, methods)) = …
and f_var_decl (Var (type, nm)) = …
and f_stmt s = match s with
   Block sl -> …
   If (e, s1, s2) -> …
… etc. ..
and f_exp e = match e with …
```