CS 421 Lecture 3: Even More OCaml

- Announcements

- Lecture outline
  - Type declaration in OCaml
  - Trees
  - Polymorphic types
  - Abstract syntax

Announcements

- Reminder: no "live" lectures next Monday & Tuesday (June 8, 9)
  - Pre-recorded lecture videos have been posted on the web site
- Reminder: limited course staff availability this weekend
  - Friday – Sunday you are on your own!
- MP2 has been posted
  - Due 1:00PM Wednesday, June 10

Brief review

- Tuples
  - Fixed-size, heterogeneous collections
  - Ex: ("hello", "cs", 421)
    - Type: string * string * int
- Pairs
  - Tuples with two values
    - fst, snd functions
- Lists
  - Variable-size, homogeneous collections
  - Ex: [1; 2; 3; 4; 5]
    - Type: int list
  - :: - cons, @ - append
    - [1; 2] @ (3 :: [4; 5]) = [1; 2; 3; 4; 5]

Brief review

- Pattern matching
  - let incr_second_of_3 (x,y,z) = y+1;;
    - Type: `a * int * `c -> int
  - let sum_pair p = (fst p) + (snd p)
    - Type: int * int -> int
- Match expressions
  - Pattern matching with choice among alternate options
  - let rec is_even lst = match lst with

```
  [] -> true
| x::[] -> false
| x::y::ys -> is_even ys
```

    - Type: `a list -> boolean

Type declaration in OCaml

- First, type expressions are:
  - te = int | string | unit | … | te list | te * te * … * te

Type declaration in OCaml

- type t = te
  - After this, t is an abbreviation for te
  - Similar to "let" syntax for names
- type t = C_1 [of te_1] | … | C_n [of te_n]
  - Where C_1, …, C_n are *constructor names* – names that 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_1, *etc.*

Example 1

- Enumerated types

```
type weekday = Mon | Tues | Wed | Thurs | Fri;;
let today = Thurs;;
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 2

- Disjoint unions

```
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 a type error!

- Corresponds to what is called *discriminated union*, *tagged union*, *disjoing union*, or *variant record*.

Example 2 (cont)

- Disjoint unions

```
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;
}

void shape_to_string(struct shape s) {
   switch (s.type_of_shape) {
   case 0: cout << "circle" << s.shape_data.radius; break;
   …
   }
}
```

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 = …;
   }
   void shape_to_string(Shape s) {
      // similar to 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 if (…)
   sh = new Square(…);
…
Sh.shape_to_string();
```

Recursive type definitions in OCaml

- In “type t = C of e | …”, e 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

- Example: Create a list of all the integers in a tree.
  - Use 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

- Similar to “let … and …” syntax

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

- Example: abstract syntax

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"))
```

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 the AST:

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