Lecture 3 — User-defined types

- In this lecture, you will learn to define trees in OCaml (analogous to what you might do in Java or C++ by defining a `tree` class). This will allow us to define *abstract syntax trees*, which we will use extensively in this class. Abstract syntax trees are the central data structure in a compiler.
- Specific topics:
  - User-defined types in OCaml
  - Defining trees (including ASTs)

Type abbreviations in OCaml

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

  $$\texttt{type}\ t = te$$

- te **is a type expression:**

  $$te = \texttt{int} \mid \texttt{float} \mid \ldots \mid te * te * \cdots * te \mid te\ \texttt{list} \mid te \rightarrow te$$

- **Examples of type expressions:**

Defining new types

- OCaml allows you to create new types by writing:

  $$\texttt{type } t \texttt{ = } C_1 \texttt{ [of } te_1\texttt{]} \texttt{ | } \ldots \texttt{ | } C_n \texttt{ [of } te_n\texttt{]}$$

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

- The above declaration creates a new type, called t, and *automatically* creates new functions that construct values of type t:
  - $C_1 : te_1 \longrightarrow t$
  - ...
  - $C_n : te_n \longrightarrow t$

Defining new types (cont.)

- For example, suppose we define this type:

```
type form_of_id = License of string
                | SScard of int * int * int
                | Student_id of string
```

- As soon as this is entered, you can enter:

```
# let myid = Student_id "123456789";;
val myid : form_of_id = Student_id "123456789"
# let hisid = SScard (123, 45, 6789);;
val hisid : form_of_id = SScard (123, 45, 6789)
```

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
```
$$\begin{array}{l} \quad C_1(x, \ldots, y) \texttt{ -> } e_1 \\ \texttt{| } C_2(x, \ldots, y) \texttt{ -> } e_2 \\ \texttt{| } \ldots \\ \texttt{| } C_n(x, \ldots, y) \texttt{ -> } e_n \end{array}$$

Type definition example

```
type form_of_id = License of string
                | SScard of int * int * int
                | Student_id of string


let string_of_id id =
    match id with
        License s -> "license " ^ s
      | SScard (x,y,z) -> "ssnum " ^ (string_of_int x)
                          ^ (string_of_int y) ^ (string_of_int z)
      | Student_id s -> "uin " ^ s
```

Type definition exercise

- **Given type**

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

define function `string_of_shape: shape → string` **that prints the shape (e.g. outputs "**`circle 4.3`**" for a circle):**

```
let string_of_shape sh =
    match sh with
      Circle r -> "circle" ^ string_of_float r
    | Square r -> "square" ^ string_of_float r
    | Triangle (x,y,z) -> "triangle" ^ string_of_float x ^ " " ^
                          string_of_float y ^ " " ^ string_of_float z;;
```

Recursive type definitions

- **In this type definition:**

$$\texttt{type } t \texttt{ = } C_1 \texttt{ [of } te_1\texttt{]} \texttt{ | } \ldots \texttt{ | } C_n \texttt{ [of } te_n\texttt{]}$$

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

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

- **Ex: write the function** `sum : mylist → int.`

```
let rec sum ls = match ls with
    Empty -> 0
  | Cons (x, xs) -> x + sum xs ;;
```

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: bintree → bool**

```
let isLeaf t = match t with
    Empty -> false
    Node(i, t1, t2) -> if t1 = Empty && t2 = Empty then true
                       else false;;
```

- **Define sum: bintree → int**

```
let sum t = match t with
    Empty -> 0
    Node(i, t1, t2) -> i + sum t1 + sum t2;;
```

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 ocamlexpr = Name of string | Intconst of int
               | Let of definition * ocamlexpr
and definition = Def of string * ocamlexpr
```

- The above defines two types and four constructors:
  - `Name: string → ocamlexpr`
  - `Intconst: int → ocamlexpr`
  - `Let: definition * ocamlexpr → ocamlexpr`
  - `Def: string * ocamlexpr → definition`

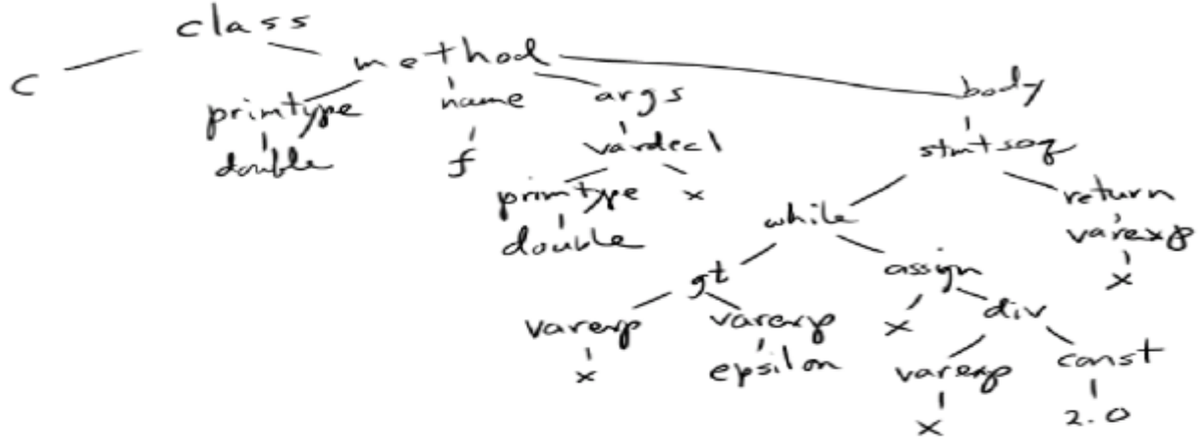
Abstract syntax

- *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.**

Ex: Abstract syntax of simple expressions

- Here is an abstract syntax for simple arithmetic expressions as an OCaml data type:

```
type expr = Int of int | Plus of expr*expr
          | Times of expr*expr | Negate of expr
```

- For example:
  - `Plus(Int 3, Int 5)` is abstract syntax for `3+5`
  - `Plus(Int 3, Times(Int 5, Int 6))` is abstract syntax for `3+5*6` or `3+(5*6)`
  - `Times(Plus(Int 3, Int 5), Int 6)` is abstract syntax for `(3+5)*6`

Exercises using expr

```
type expr = Int of int | Plus of expr*expr
          | Times of expr*expr | Negate of expr
```

- **Show the abstract syntax tree for expression** `4+-(7*-8+4)`:

- **Give the OCaml expression of type** `expr` **for that tree:**

Plus (Int(4), Negate (Times (Int(7), Negate (Plus (Int(8), Int(4))))))

Exercises using `expr` (cont.)

- **Write the function** `countPluses: expr → int`, **which counts the number of** `Plus` **operations in an** `expr`:

```
type expr = Int of int | Plus of expr*expr
          | Times of expr*expr | Negate of expr

let rec countPluses e = match e with
  Int i -> 0
| Plus(e1, e2) -> 1 + countPluses e1 + countPluses e2
| Times(e1, e2) -> countPluses e1 + countPluses e2
| Negate e -> countPluses e ;;
```

Exercises using `expr` (cont.)

- **Write the function `eval: expr → int`, which evaluates its argument, e.g. `eval (Times(Negate(Int 5), Int 6)) = -30`.**

```
let rec eval e = match e with
  Int i ->  i
| Plus(e1, e2) ->  eval e1 + eval e2
| Times(e1, e2) ->  eval e1 * eval e2
| Negate e ->  ~ eval e
```

Wrap-up

- Today we discussed:
  - How to define new types in OCaml
  - Especially trees
  - Especially abstract syntax trees
- We discussed it because:
  - ASTs are central to writing compilers
- In the next class, we will:
  - Do more programming with ASTs
- What to do now:
  - Just come back on Thursday...