Benjamin Fenster
CS-103 (Skalka)
5 October 2003

Type Declarations

I. Type Declarations as Abbreviations
   a. `type numeric = int`
   b. `type intStack = int list`
   c. `type student = string * int * float (* name, ssn, GPA *)`
   d. Parameterized by Type Variables
      i. `type 'a stack = 'a list`
      ii. NB: Any type variable appearing on the right side of a type declaration must appear also on the left side (that is, must be bound)
      iii. type stack = ‘a list (NO!)

II. Variants
   a. Abbreviations don’t add anything NEW. Variants give truly new subjects for the program.
   b. Can take different shapes (thus “Variants”)
   c. Example: Binary Trees
      i. `type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree`
      ii. NB: This is a recursive type definition
      iii. Leaf has no value – a leaf is really at the edge (like the NULL pointer in C++)
      iv. `Node(Node(Leaf, 2, Leaf), 1, Leaf(Node(Leaf, 6, Leaf), 5, Leaf))`
   d. Deconstruction
      i. Pattern matching
      ii. The patterns are defined entirely by the types.
      iii. `let rec inorder t = match t with Leaf -> [] | Node(tl, v, tr) -> (inorder tl) @ [v] @ (inorder tr) : 'a tree -> 'a list`
      iv. NB: @ is list append
   e. Recursive vs. Non-Recursive Datatypes
      i. Non-Recursive Definitions
         1. `type 'a option = None | Some of 'a`
         2. Used when a function may or may not return a value.
         3. `let f x = if (p(x) then Some(x + 1) else None`
      ii. Recursive
         1. Don’t forget to have a basis for a recursive datatype!
         2. `type circular = Circ of Circular`
         3. In some languages this definition is meaningful, but not OCaml
   f. Capturing the behavior of a tree
      i. `let treefold basis step tree = match t with Leaf -> basis | Node(lt, v, rt) -> step(v, treefold basis step lt, treefold basis step tr) : 'a -> (('b * 'a * 'a) -> 'a) -> 'b tree -> 'a`
      ii. `let inorder = treefold [] (fun (v, lt, rt) -> lt @ [v] @ rt) : 'a tree -> 'a list`
      iii. `let preorder = treefold [] (fun (v, lt, rt) -> [v] @ lt @ rt) : 'a tree -> 'a list`

III. Records
   a. Like structs in C/C++
   b. Collections of named values.
   c. The difference is that, per the functional programming norm, fields are immutable.
   d. Example
      i. `type student = { name : string ; email : string; gpa : float }`

ii. `let bob = {name = “bob”; email = “bob@zoo.uvm.edu”; gpa = 3.6 }`
iii. Order in which field are given values is arbitrary
iv. Cannot partially define the type. Must give a value for all fields.
v. Field names are unique! type xyz = { name : string; } will replace the old type!
vi. bob.gpa $\Downarrow$ 3.6

e. Formalities
   i. Given `type r =` $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$
   ii. Typing Rules
      1. $\{l_1 = e_1; \ldots; l_n = e_n\} : r$ iff $\forall\ 1 \le i \le n,\ e_i : \tau_I$
      2. $e_i l_i : \tau_I$ iff $e : r$ and $1 \le i \le n$
   iii. Evaluation
      1. $\{l_1 = e_1; \ldots; l_n = e_n\} \Downarrow \{l_1 = v_1; \ldots; l_n = v_n\}$ iff $\forall\ 1 \le i \le n$ and $e_1 \Downarrow v_i$ evaluated in right-to-left order
      2. $e_1 l_1 \Downarrow v_1$ iff $e \Downarrow \{l_1 = v_1; \ldots; l_n = v_n\}$ and $1 \le i \le n$
   iv. Pattern matching on records
      1. `type r = {a : int; b : int}`
      2. `let project_a {a = x; b = _} = x`
      3. `type r = {a : int * float; b = int}`
      4. `let project_a1 = {a = (x, _ ); b = _ } = x`

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: