Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
Recursion over Recursive Data Types

# type exp = VarExp of string | ConstExp of const
| BinOpAppExp of bin_op * exp * exp
| FunExp of string * exp | AppExp of exp * exp

- How to count the number of variables in an exp?
Recursion over Recursive Data Types

# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp

- How to count the number of variables in an exp?

# let rec varCnt exp =
    match exp with VarExp x ->
      | ConstExp c ->
      | BinOpAppExp (b, e1, e2) ->
      | FunExp (x,e) ->
      | AppExp (e1, e2) ->
Recursion over Recursive Data Types

# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp

- How to count the number of variables in an exp?

# let rec varCnt exp =
  match exp with
  VarExp x -> 1
  ConstExp c -> 0
  BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
  FunExp (x, e) -> 1 + varCnt e
  AppExp (e1, e2) -> varCnt e1 + varCnt e2
Your turn now

Try Problem 3 on MP3
Mutually Recursive Types

```ocaml
# type 'a tree = TreeLeaf of 'a
  | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree
  | More of ('a tree * 'a treeList);

type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList)
```
Mutually Recursive Types - Values

# let tree =
TreeNode
  (More (TreeLeaf 5,
    (More (TreeNode
      (More (TreeLeaf 3,
        Last (TreeLeaf 2))),
    Last (TreeLeaf 7)))));;
val tree : int tree =
  TreeNode
  (More
    (TreeLeaf 5,
      More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))), Last (TreeLeaf 7))))
Mutually Recursive Types - Values

TreeNode
  More
    TreeLeaf
      5

TreeNode
  More
    TreeLeaf
      3
  Last
    TreeLeaf
      2
Mutually Recursive Types - Values

A more conventional picture

```
      5
     /|
    / \
   3   2
```

```
      7
```

 Mutually Recursive Functions

```ocaml
# let rec fringe tree = 
  match tree with (TreeLeaf x) -> [x]
| (TreeNode list) -> list_fringe list
and list_fringe tree_list = 
  match tree_list with (Last tree) -> fringe tree
| (More (tree,list)) ->
  (fringe tree) @ (list_fringe list);;

val fringe : 'a tree -> 'a list = <fun>
val list_fringe : 'a treeList -> 'a list = <fun>
```
Mutually Recursive Functions

# fringe tree;;
- : int list = [5; 3; 2; 7]
Problem

# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size
Problem

```ocaml
# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size
let rec tree_size t =
    match t with TreeLeaf _ ->
    | TreeNode ts ->
```
Problem

# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size

let rec tree_size t =
  match t with
  TreeLeaf _ -> 1
| TreeNode ts -> treeList_size ts
# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size and treeList_size

let rec tree_size t =
  match t with
  | TreeLeaf _ -> 1
  | TreeNode ts -> treeList_size ts
and treeList_size ts =
# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size and treeList_size

let rec tree_size t =
    match t with TreeLeaf _ -> 1
    | TreeNode ts -> treeList_size ts
and treeList_size ts =
    match ts with Last t ->
    | More t ts' ->
Problem

# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size and treeList_size

let rec tree_size t =
    match t with TreeLeaf _ -> 1
    | TreeNode ts -> treeList_size ts
    | TreeLeaf _ -> 1

and treeList_size ts =
    match ts with Last t -> tree_size t
    | More t ts' -> tree_size t + treeList_size ts'
Problem

# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size and treeList_size

let rec tree_size t =
    match t with TreeLeaf _ -> 1
    | TreeNode ts -> treeList_size ts

and treeList_size ts =
    match ts with Last t -> tree_size t
    | More t ts' -> tree_size t + treeList_size ts'
Nested Recursive Types

# type 'a labeled_tree =
TreeNode of ('a * 'a labeled_tree list);

type 'a labeled_tree = TreeNode of ('a * 'a labeled_tree list)
Nested Recursive Type Values

# let ltree =
      TreeNode(5,
              [TreeNode (3, []);
               TreeNode (2, [TreeNode (1, []);
                              TreeNode (7, [])]);
               TreeNode (5, [])]);
val ltree : int labeled_tree =
TreeNode
(5,
  [TreeNode (3, []); TreeNode (2, [TreeNode (1, []); TreeNode (7, []))];
TreeNode (5, []))
Nested Recursive Type Values

Ltree = TreeNode(5)

TreeNode(3)   TreeNode(2)   TreeNode(5)

TreeNode(1)   TreeNode(7)
Nested Recursive Type Values
Mutually Recursive Functions

```ocaml
# let rec flatten_tree labtree = match labtree with
   | TreeNode (x, treelist) -> x :: flatten_tree_list treelist
and flatten_tree_list treelist = match treelist with
   | [] -> []
   | labtree :: labtrees -> flatten_tree labtree @ flatten_tree_list labtrees;;
```
Mutually Recursive Functions

val flatten_tree : 'a labeled_tree -> 'a list = <fun>
val flatten_tree_list : 'a labeled_tree_list -> 'a list = <fun>

# flatten_tree ltree;
- : int list = [5; 3; 2; 1; 7; 5]

- Nested recursive types lead to mutually recursive functions
Infinite Recursive Values

```ocaml
# let rec ones = 1::ones;;
val ones : int list =
  [1; 1; 1; 1; ...]
# match ones with x::_ -> x;;
Characters 0-25:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  match ones with x::_ -> x;;
    ~~~~~~~~~~~~~~~~~~~~~~~~~
- : int = 1
```
Infinite Recursive Values

```ml
# let rec lab_tree = TreeNode(2, tree_list)
    and tree_list = [lab_tree; lab_tree];;
val lab_tree : int labeled_tree = 
    TreeNode (2, [TreeNode(...); TreeNode(...)])
val tree_list : int labeled_tree list = 
    [TreeNode (2, [TreeNode(...); TreeNode(...)]);
     TreeNode (2, [TreeNode(...); TreeNode(...)])]
```
Infinite Recursive Values

# match lab_tree

  with TreeNode (x, _) -> x;;

- : int = 2
Records

- Records serve the same programming purpose as tuples.
- Provide better documentation, more readable code.
- Allow components to be accessed by label instead of position.
  - Labels (aka *field names* must be unique).
  - Fields accessed by suffix dot notation.
Record Types

- Record types must be declared before they can be used in OCaml

```ocaml
# type person = {name : string; ss : (int * int * int); age : int};;
type person = { name : string; ss : int * int * int; age : int; }
```

- person is the type being introduced
- name, ss and age are the labels, or fields
Record Values

- Records built with labels; order does not matter

```ocaml
# let teacher = {name = "Elsa L. Gunter";
    age = 102; ss = (119,73,6244)};;
val teacher : person =
    {name = "Elsa L. Gunter"; ss = (119, 73, 6244); age = 102}
```
Record Pattern Matching

```ocaml
# let {name = elsa; age = age; ss = (_,_,s3)} = teacher;;
val elsa : string = "Elsa L. Gunter"
val age : int = 102
val s3 : int = 6244
```
Record Field Access

```ocaml
# let soc_sec = teacher.ss;;

val soc_sec : int * int * int = (119, 73, 6244)
```
# let student = {ss=(325,40,1276); name="Joseph Martins"; age=22};;
val student : person = 
{name = "Joseph Martins"; ss = (325, 40, 1276); age = 22}

# student = teacher;;
- : bool = false
New Records from Old

# let birthday person = {person with age = person.age + 1};;
val birthday : person -> person = <fun>

# birthday teacher;;

- : person = {name = "Elsa L. Gunter"; ss = (119, 73, 6244); age = 103}
New Records from Old

# let new_id name soc_sec person =
    {person with name = name; ss = soc_sec};
val new_id : string -> int * int * int -> person
    -> person = <fun>
# new_id "Guieseppe Martin" (523,04,6712) student;

- : person = {name = "Guieseppe Martin"; ss
    = (523, 4, 6712); age = 22}