Data structures

Ziyan Maraikar

September 2, 2014

LECTURE OUTLINE

DATA STRUCTURES AND ALGORITHMS

★ Data structures are containers that store our data in a particular fashion,
  e.g., lists store data as a *sequence* of items.

★ We can build *algorithms* that do useful operations on data structure,
  e.g., search for an item, or list items in sorted order.

★ The data structure determines what algorithms can be *efficiently* implemented,
  e.g., how quickly can we search and sorted items.

CONSTRUCTION VS. TRAVERSAL

|	Construction	Traversal
Example	```ocaml\ntype 'a list =\n	Empty\n	Cons of 'a * 'a\n list\n```	```ocaml\nlet copy list =\n	Empty -> Empty\n	Cons(hd, tl) ->\n Cons(hd, copy tl)\n```
Inductive	builds up the DS recursively	*traverse* the DS item by item
Terminal	marks the end of the DS	stop the traversal

LECTURE OUTLINE

Trees

★ A *tree* stores items in a hierarchical structure. Useful for storing data in a particular order for fast searching and sorting.

★ Each *node* in the tree has exactly one *parent* above it and zero or more *children* below it.

★ The topmost node is called the *root* and the nodes at the bottom are called *leaves*.

EXAMPLE
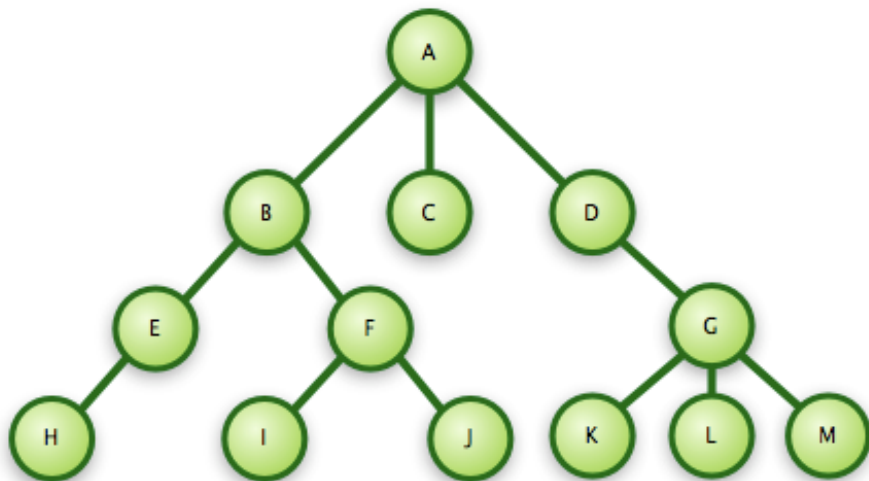
Binary trees

In a *binary tree* a node has two children.

We define a binary tree inductively as either,

★ Empty

★ Consisting of a **root node with two chlidren that are binary trees**.

```
type 'a binarytree =
| Empty
| Node of 'a * 'a binarytree * 'a binarytree
```

Example:

```
let t1 = Node (3, Empty, Node (2, Empty, Empty))
```

TREE TRAVERSAL

We often need to *traverse* a tree visiting each node.

In what order should we visit the nodes? There are three possibilities

PREORDER Visit the root, then the left subtree, finally the right subtree.

INORDER Visit the left subtree, then the root, finally the right subtree.

POSTORDER Visit the the left subtree, then the right subtree, finally the root.

PREORDER TRAVERSAL

```ocaml
type 'a binarytree =
  | Empty
  | Node of 'a * 'a binarytree * 'a binarytree

let rec preorder t =
  match t with
  | Node(o, l, r) -> string_of_int o ^ ","
                     ^ preorder l ^ preorder r
  | Empty -> ""
```

EXERCISE

★ Define a function `tree_square` that takes an `int tree` and squares each node.

★ Define a function `tree_sum` that takes an `int tree` and returns the sum of all nodes.

★ Define a function `flatten_inorder` that takes a `tree` and returns an inorder list of nodes.

LECTURE OUTLINE

MAP ON TREES

We already saw the `map` operation on lists

```
(* Apply the function f to each element of list l *)
let map f l =
  match l with
  hd::tl -> f hd :: map f tl
  [] -> []
```

This is an example of a tree map.

```
let rec tree_square t =
  match t with
  | Node(o, l, r) -> Node(square o, tree_square l,
    tree_square r)
  | Empty -> Empty
```

Can we generalise this to a map operation on trees?

MAP ON TREES

Two cases to consider:

> EMPTY just return `Empty`.
>> NODE apply `f` to value at node, and map `f` on left and right subtrees.

```
let rec tree_map f t =
  match t with
  | Empty -> Empty
  | Node(v, l, r) -> Node(f v, tree_map f l, tree_map f r)
```

FOLD ON TREES

We already saw the `fold` operation on lists

```
(* Compose elements of l using function f and identity e *)
let fold f e l =
  match l with
  hd::tl -> f hd (fold f e tl)
  [] -> e
```

This is an example of a tree fold.

```
let rec tree_sum t =
  match t with
  | Node(o, l, r) -> o + tree_sum l + tree_sum r
  | Empty -> 0
```

Fold on trees

Empty return identity.

Node fold f on left and right subtrees and combine with value at node.

We need to combine *three* values togther using a *binary* function. In what order should we combine the values using f?

```
let rec tree_fold f e t =
  match t with
  | Empty -> e
  | Node(v, l, r) ->
    let lv = tree_fold f e l in
    let lr = tree_fold f e r in
    f v (f lv lr)
```

If f is *associative* and *commutative* the order in which we apply f will not matter.

DATA STRUCTURES AND ALGORITHMS

★ Define a function `tree_square` using `tree_map`.

★ Define `tree_sum` using `tree_fold`.

LECTURE OUTLINE

Arithmetic expressions

Consider the structure of arithmetic expressions

$$(1 + 2) \times 3$$

How could we represent them in Ocaml?

Arithmetic expressions

We can define an arithmetic expression recursively as either,

★ a constant (number), or

★ arithmetic expression(s) composed using an operator.

We can turn this into a type definition.

```
type arexp =
| Const of int
| Plus of arexp * arexp
```

What does this definition look like?

EXERCISE

```
type arexp =
| Const of int
| Plus of arexp * arexp
```

★ Add the `Times` operator to this type.

★ Construct the expression $(1 + 2) \times 3$

Evaluating an expression

```
type arexp =
  | Const of int
  | Plus of arexp * arexp
  | Times of arexp * arexp
```

We can use Ocaml's arithmetic operators to evaluate expression.

```
let rec eval_expr e =
  match e with
  | Const a -> a
  | Plus(a, b) -> _____ + _____
  | Times(a, b) ->  _____ * _____
```

Does this pattern look familiar? This is a (variation of) our binary tree definition. Here, the leaf (`Const`) nodes have a value associated with them.

EVALUATING AN EXPRESSION

```
let rec eval_expr e =
  match e with
  | Const a -> a
  | Plus(a, b) -> eval_expr a + eval_expr b
  | Times(a, b) -> eval_expr a * eval_expr b
```

This is an inorder traversal of the expression tree!

PRINTING AN EXPRESSION

Exercise: Write a `pretty_print` function that converts an expression to a string.

```
let rec pretty_print e =
  match e with
  | Const a -> pretty_print a
  | Plus(a, b) -> pretty_print a ^ " + " ^ pretty_print b
  | Times(a, b) -> pretty_print a ^ " * " ^ pretty_print b
```

THE EXPRESSION PROBLEM

This is a good problem to study how software systems can be extended in different dimensions.

1. How to add new operators to expressions without modifying existing code.
   e.g. Subtraction, division.

2. How to add new functions that operate on expressions without modifying existing code.
   e.g. pretty printing, evaluation.

The study of how to accomplish this is called the *expression problem.*

In which dimension is our solution extesible?

Lecture Outline