CS3110 Fall 2013 Lecture 9: Environment Model (9/26)

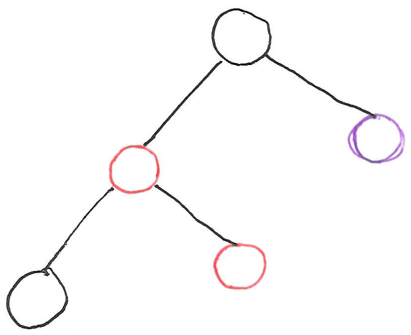
Robert Constable

1 Lecture Plan

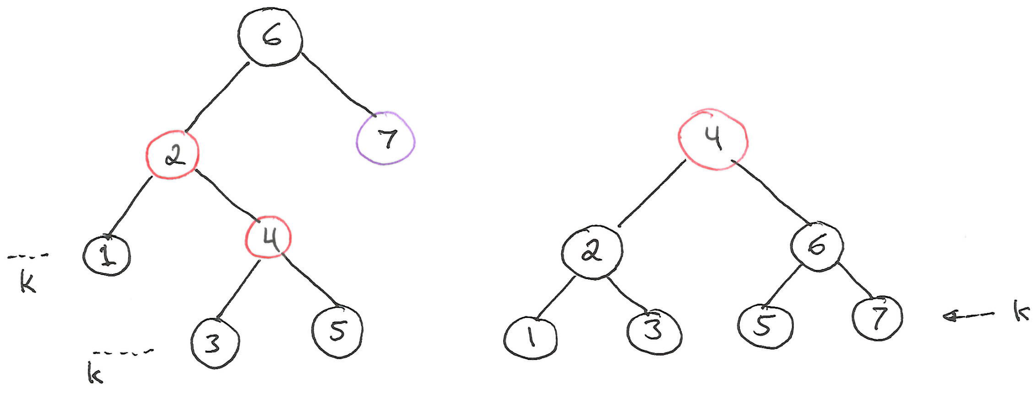
1. Comments on Red-Black tree insert operation (See Fall 2009 CS3110 Lecture 10 posted as Lectures 7, 8 on our web page.)
2. Comments on understanding, coding and proving complex algorithmic problems – methodology
3. The environment model needed for PS3 (See Spring 2011 Lecture 12)
4. Comments on BNF and Variant Types (See Fall 2009 Lecture 4)

2 Red Black Tree – insert "fix-up stage"

We insert as in binary search trees and we color the node **red** (this preserves the invariant that each path to a leaf has the same number of black nodes). We might end up with a tree like this:



We need to rebalance and preserve the black height. Number this subtree to have the bst property.



See page 4, Lecture 10, Spring 2011.

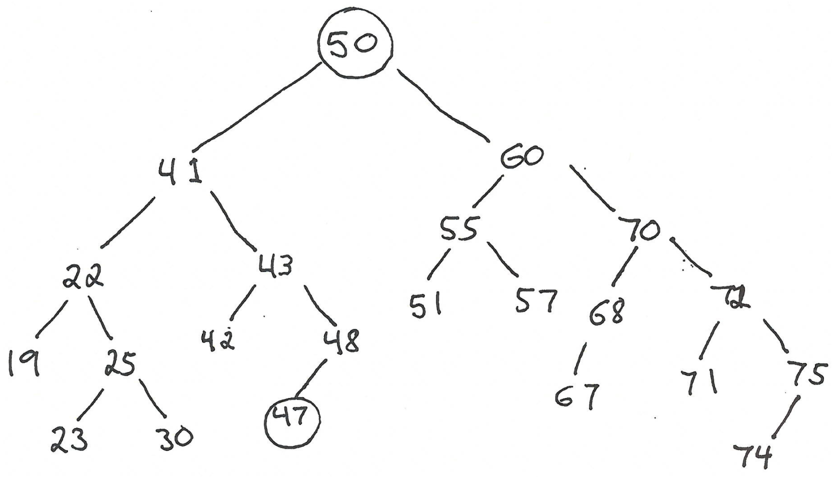
3 Comments on methodology

- create simple examples of the data structure either "by hand" or by building the constructors first, e.g. like my binary search tree from Lecture 6
- build the easy functions for "experimenting" with the trees, e.g. insert, find, max, delete_raw, traverse
- build operations for specification, e.g. mth, length; for trees build **depth**, path to node, index by path, etc.
- try to structure an argument or assemble evidence as your code executes

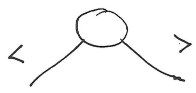
Mike gave a virtuoso demonstration of how the structural induction style informal proof for red black insertion drove the code.

4 Binary search trees (BST's)

Here is one



We need a type such as *int* with a **linear order** relation $<$. The tree has the property that at each node $\textcircled{n}$ the **left subtree** has elements **less than n** and the right subtree has elements greater.



```
val example2 : int_bst =
  Node
   (50,
    Node
     (41,
      Node
       (22, Node (19, Nil, Nil),
        Node (25, Node (23, Nil, Nil), Node (30, Nil, Nil))),
      Node (43, Node (42, Nil, Nil), Node (48, Node (47, Nil, Nil), Nil))),
    Node
     (60, Node (55, Node (51, Nil, Nil), Node (57, Nil, Nil)),
      Node
       (70, Node (68, Node (67, Nil, Nil), Nil),
        Node (72, Node (71, Nil, Nil), Node (75, Node (74, Nil, Nil), Nil)))))
```

5 The environment model of evaluation

Prof. Kozen discusses this topic in Lecture 12 motivating by saying that:

> The substitution model is inefficient and the environment model is not and more closely models how the OCaml interpreter actually works.
>
> He covered the substitution model in Lecture 6. We covered that as well with our account of reduction rules.

A key idea needed for the environment model is the notion of the **scope** of a variable binding as in

let statements

$let\ f\ x = e_1\ in\ e_2$ $\qquad\qquad$ $let\ x = 2\ and\ y = 3\ in\ x + y$

in **function bodies**

$fun\ x \rightarrow fun\ y \rightarrow fun\ x \rightarrow x + y$

5.1 Scope and binding in "lets" and functions

Notice that simple *let* statements can be expressed as function application.

$let\ x = e_1\ in\ e_2$	is just	$(fun\ x \rightarrow e_2)\ e_1$	clearly
$let\ x = e_1\ in\ e_2(x)$	is just	$(fun\ x \rightarrow e_2(x))\ e_1$	$\downarrow \quad e_2(e_1/x)$

This looks more compact with lambda notation

$let\ x = e_1\ in\ e_2(x) \qquad$ is $\qquad (\lambda x.\, e_2(x))\ e_1.$

We can convert most **let statements** to **these function applications**, e.g.

> $let\ f\ x = 2 * x\ in\ f\ 3$
> we know $f\ 3\ \downarrow 6$
> this is also $let\ f = \lambda x.\, 2 * x$
> in $f\ 3$, we get
> $\quad f\ 3 \downarrow 6$

What is the **scope** of the x binding in $fun\ x \rightarrow exp$? e.g.

(a) $fun\ x \rightarrow fun\ y \rightarrow x * y$ versus

(b) $fun\ x \rightarrow fun\ y \rightarrow (fun\ x \rightarrow x * x)$ versus

(c) $fun\ x \rightarrow fun\ y \rightarrow fun\ z \rightarrow x * y * z$

This is a bit more compact with λ-notation

(a) $\lambda x.\, \lambda y.\, x * y$

(b) $\lambda x.\, \lambda y.\, \lambda x.\, x * x$

(c) $\lambda x.\, \lambda y.\, \lambda z.\, x * y * z$

In (a) the scope of λx is $\lambda y.\, x * y$ and the scope of λy is $x * y$.
In (b) the scope of the first λx is $\lambda y.\, \lambda x.\, x * x$, but it is the inner most occurrence of λx that **binds** x.

In PS3 we discuss **lazy evaluation** in which the argument to a function might not be a canonical value but an expression that needs to be reduced. For example suppose we have introduced a *let* environment as follows:

```
(* NORMAL USE OF LET, NO CAPTURE, EAGER EVALUATION *)

# let x = 5 in (fun y -> (fun x -> y * x) x ) 3 ;;
- : int = 15


(* CAPTURE OF let x = 5 BY INNER (fun x -> y * x)  *)

# let x = 5  in (fun y -> (fun x -> y * x) ) 3 ;;
Characters 4-5:
  let x = 5  in (fun y -> (fun x -> y * x) ) 3 ;;
      ^
Warning 26: unused variable x.
- : int -> int = <fun>
```

What happened is that $fun\ x \rightarrow y * x$ "**captured the x**" and ignored the $let\ x = 5$ binding.

5.2 Lazy evaluation

We won't study lazy evaluation in OCaml, although there is a package that supports it. These remarks are meant as background for PS3 and as background for a broader discussion of programming language design. Haskell and Nuprl use lazy evaluation. In Nuprl it is the default but eager evaluation is also used, as "call by value."

In lazy evaluation, how would we solve the problem of capture? Note, we don't want to force the evaluation of x to 5 until we need to evaluate it.

Capture is avoided by renaming the binding variable that is doing the "capturing," so we rewrite $fun\ x \to x * x$ to $fun\ x1 \to x * x1$ (this is called α-conversion). Now we get

$$\begin{array}{cl} (fun\ x1 \to x * x1)\ 3 & \\ \downarrow & \\ x * 3 & \quad \text{where } x = 5 \text{, hence} \\ \downarrow & \\ 15 & \end{array}$$

5.3 Binding and scope

So in the presence of lazy evaluation we must be especially careful about scope and binding.

But even for OCaml's call by value semantics, we must be clear about the evaluation environment. In the environment model, we avoid substitution until a value is needed by keeping track of variable bindings in a list called a **closure**.

We will illustrate the use of closures in an example and define them more fully next time. Meanwhile **please read** Prof. Kozen's **Lecture 12, Spring 2011**. Also look at **Recitation 6** Spring 2011, at the **evil example** page 3.

5.4 Evaluation by substitution example

λ notation version

$$(\lambda y.\,(\lambda u.\,\lambda y.\,u\;4)\;\lambda x.\,y)\;3\;\;2$$

$$\downarrow$$

$$(\lambda u.\,\lambda y.\,u\;4)\;\underline{\lambda x.\,3}\;\;2$$

$$\downarrow$$

$$\lambda y.\,(\lambda x.\,3)\;4\;\;2$$

$$\downarrow$$

$$(\lambda x.\,3)\;2$$

$$\downarrow$$

$$3$$

5.5 Evaluation by closures

function	**closure**
$(\lambda y.\,(\lambda u.\,\lambda y.\,u\;4)\;\lambda x.\,y)\;3\;\;2$	$[\;]$
$\downarrow$	
$(\lambda u.\,\lambda y.\,u\;4)\;\lambda x.\,y\;\;2$	$[y=3]$
$\downarrow$	
$(\lambda y.\,u\;4)\;\;2$	$[y=3,\;u=\{\lambda x.\,y,\;[y=3]\}]$
$\downarrow$	
$u\;4$	$[y=2,\;u=$ " " $]$
$\downarrow$	
$\{\lambda x.\,y,\;[y=3]\}\;4$	[" " "]
$\downarrow$	
$\lambda x.\,y\;4$	[$y=3$]
$\downarrow$	
y	$[y=3,\;x=4]$
$\downarrow$	
3	

Abstract syntax and variant types

Earlier we noticed that there is a similarity between BNF declarations and variant type declarations. In fact, we can define variant types that act like the corresponding BNF declarations. The values of these variant types then represent legal expressions that can occur in the language. For example, consider a BNF definition of legal OCaml type expressions:

(base types)	b ::= `int` | `float` | `string` | `bool` | `char`
(types)	t ::= b | t `->` t | t_1 `*` t_2 `*`...`*` t_n | `{` x_1 `:` t_1 `;` ... `;` x_n `:` t_n `}` | X

This grammar has exactly the same structure as the following type declarations:

```
type id = string
type baseType = Int | Real | String | Bool | Char
type mlType = Base of baseType | Arrow of mlType*mlType
            | Product of mlType list | Record of (id * mlType) list
            | DatatypeName of id
```

Any legal OCaml type expression can be represented by a value of type `Type` that contains all the information of the corresponding type expression. This value is known as the *abstract syntax* for that expression. It is abstract because it doesn't contain any information about the actual symbols used to represent the expression in the program. For example, the abstract syntax for the expression `int * bool -> {name : string}` would be:

```
Arrow (Product (Cons (Base Int, Cons (Base Bool, Nil))),
       Record (Cons (("name", Base String), Nil)))
```

The abstract syntax would be exactly the same even for a more verbose version of the same type expression: `((int * bool) -> {name : string})`. Compilers typically use abstract