Functional and Logic Programming

Lecture 10 — Trees

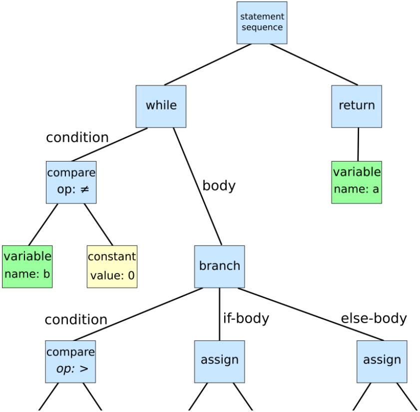
Pedro Vasconcelos
DCC/FCUP

2025

Contents

Trees

Tree structures are often used for searching or organizing information.



Unlike nature, trees in computer science grow downwards!

Contents

Binary trees

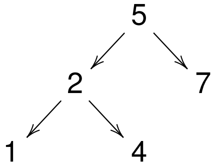
- Trees may have nodes with zero or more children
- Search trees are usually binary: nodes have exacly two children
- A value of a binary tree can be of two forms:

  a node with some payload and two children;

  a leave a terminal with no children.

Recursive definition

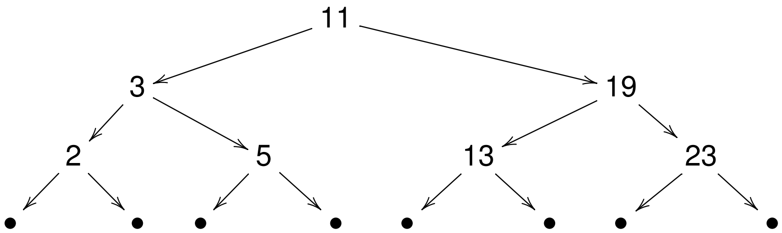
```
data Tree a    -- tree of 'a's
     = Leaf    -- terminal
     | Node a (Tree a) (Tree a)
     -- payload, left subtree and right subtree

example :: Tree Int
example
   = Node 5
     (Node 2
        (Node 1 Leaf Leaf)
        (Node 4 Leaf Leaf))
     (Node 7 Leaf Leaf)
```



Search trees

A binary tree is ordered (aka a binary search tree) if the value at each node is:

- greater than the values of the left subtree;
- smaller than the values of the right subtree.



Sets

Let us use binary search trees to implement an abstract type for sets.[1]

```
data Set a          -- set of 'a's

-- "smart" constructors
empty :: Set a
fromList :: Ord a ⇒ [a] → Set a

-- operations
-- check membership
member :: Ord a ⇒ a → Set a → Bool
-- add an element
insert :: Ord a ⇒ a → Set a → Set a
```

[1] A simplified version of `Data.Set`.

Implementation

We define a module that exports only the type name and the operations; the implementation is encapsulated in the module.

```
module Set (Set,
            empty, fromList,
            member, insert) where

data Set a = Empty
           | Node a (Set a) (Set a)
...
```

Let use see the implementation of the smart constructors and operations.

Empty set

The empty set is the single Empty constructor.

```
empty :: Set a
empty = Empty
```

Checking membership

The `member` function is defined by structural recursion over tree, using comparison to go left or right at each level.

```
member :: Ord a ⇒ a → Set a → Bool
member _ Empty = False
member x (Node y left right)
    | x==y      = True
    | x<y       = member x left
    | otherwise = member x right
```

Checking membership (cont.)

Observe the type:

```
member :: Ord a ⇒ a → Set a → Bool
```

The class restriction "`Ord a =>`" restricts the type `a` of set elements to have a total order (<=, >, etc.).

Note that `Ord` is subclass of `Eq`, hence this also implies an equality comparision (==).

Inserting an element

```
insert :: Ord a ⇒ a → Set a → Bool
insert x Empty = Node x Empty Empty
insert x (Node y left right)
    | x == y = Node y left right
    | x < y  = Node y (insert x left) right
    | otherwise = Node y left (insert x right)
```

Building a set from a list

- We could repeatedly insert elements starting with an empty set:

```
fromList :: Ord a ⇒ [a] → Set a
fromList = foldr insert Empty
```

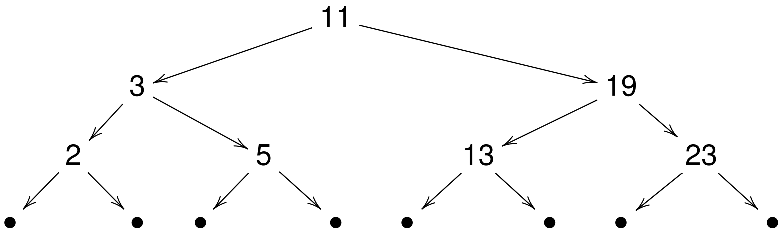
- However, this could result in a tree that is is very unbalanced e.g. if the list is sorted
- A better idea: first sort the list and then build a tree using a binary partition method (next slide)
- This ensures that the tree has minimal height and that searching takes $O(\log n)$ steps in worst case

Building a set from a list (cont.)

```
import Data.List (sort)

fromList :: Ord a ⇒ [a] → Set a
fromList xs = build (sort xs)
  where
  build [] = Empty
  build xs = Node x (build xs') (build xs'')
    where xs' = take k xs
          x:xs'' = drop k xs
          k = length xs `div` 2
```

Example

```
fromList [3,2,13,5,23,11,19]
=
Node 11
  (Node 3  (Node 2 Empty Empty)
           (Node 5 Empty Empty))
  (Node 19 (Node 13 Empty Empty)
           (Node 23 Empty Empty))
```



Contents

Syntax trees

- The syntax of formal language is frequently recursive:
  - algebraic expresssions
  - programming languages
  - HTML, XML, JSON, etc.. . .
- We can use trees to represent syntax terms
- This makes it easier to process the languages:
  - interpreters and compilers;
  - automatic formaters;
  - static analysers (type checker, "linters", etc.).

Propositions

Let us define a recursive type to represent logic propositions made up from

- constants (True, False);
- variables ($x, y, p, q, \ldots$)

and the following connectives:

- negation ($\neg$)
- conjunction ($\wedge$);
- implications ($\Longrightarrow$).

Propositions (cont.)

```
type Name = Char -- 'x', 'y', 'z', etc.
data Prop = Const Bool
          | Var Name
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

The type definition follows a context free grammar:

$$
\begin{array}{lll}
\mathit{Prop} & \longrightarrow & \text{constant} \\
 & | & \text{variable} \\
 & | & \neg \mathit{Prop} \\
 & | & \mathit{Prop} \wedge \mathit{Prop} \\
 & | & \mathit{Prop} \implies \mathit{Prop}
\end{array}
$$

Examples

$p \wedge (\neg p)$	`And (Var 'p') (Not (Var 'p'))`
$(x \wedge y) \implies y$	`Imply (And (Var 'x') (Var 'y')) (Var 'y')`
$x \wedge (y \implies y)$	`And (Var 'x') (Imply (Var 'y') (Var 'y'))`

```
Imply
    (And (Var 'x') (Var 'y'))
    (Var 'y')
```

```
And
    (Var 'x')
    (Imply (Var 'y') (Var 'y'))
```

What can we do with a syntax tree?

- Parsing: convert text into a syntax tree

  `"x && ~y"`

  $\longrightarrow$ `And (Var 'x') (Not (Var 'y'))`
- Pretty-printing: convert a syntax tree into formated text

  `And (Var 'x') (Not (Var 'y'))`

  $\longrightarrow$ `"x&&~y"`
- Interpret: walk over the tree and compute some result

  `And (Const True) (Not (Const False))`

  $\longrightarrow$ `True`
- Compile: translate the tree into a program in another language (e.g. *assembly*)

We will see examples for an interpreter and pretty-printer and leave parsing to its own lecture.

Interpreter

- Let us define a recursive `eval` function to compute the value of a proposition
- Since the value may depend on the value of variables, we need an extra argument called an *enviroment*
- We will use a simple list of pairs:

```
type Name = Char
type Env = [(Name, Bool)]
 -- associations of names to values
```

- Alternatively, we could use a “finite map” data structure e.g. `Data.Map` with more efficient search

Interpreter (cont.)

```
eval :: Env → Prop → Bool
eval env (Const b) = b
eval env (Var x)
   = case lookup x env of
          Just b → b
          Nothing → error "undefined variable"
eval env (Not p) = not (eval env p)
eval env (And p q)
     = eval env p && eval env q
eval env (Imply p q)
     = not (eval env p) || eval env q
```

The `lookup` function comes from the Prelude:

```
lookup :: Eq a ⇒ a → [(a,b)] → Maybe b
```

Pretty-printer

- A recursive function `pretty` that converts a proposition into a human-readable string
- For simplicity, we will always introduce parentheses at each logic connective

Pretty-printer (cont.)

```
pretty :: Prop -> String
pretty (Const b) = show b
pretty (Var x)   = [x]
pretty (Not p)
    = "(~" ++ pretty p ++ ")"
pretty (And p q)
    = "(" ++ pretty p ++ "&&" ++ pretty q ++ ")"
pretty (Imply p q)
    = "(" ++ pretty p ++ "=>" ++ pretty q ++ ")"
```

Efficiency

- The `pretty` function as written is inefficient because we use `++` over the result of the recursive call:

```
pretty (And p q)
    = "(" ++ pretty p ++ "&&" ++ pretty q ++ ")"
                      ^^       ^^           ^^
```

- If `pretty p` has length n then `++` requires $O(n)$ steps
- Hence `pretty` as a whole need $O(n^2)$ steps for a tree of size n
- Can we do better?
- Yes: we can make the append disappear by using and extra acumulator parameter
- The resulting `pretty` will have complexity $O(n)$

Efficiency (cont.)

```
pretty :: Prop -> String
pretty p = prettyAcc p ""

prettyAcc :: Prop -> String -> String
prettyAcc (Const b) acc = show b ++ acc
prettyAcc (Var x) acc   = [x] ++ acc
prettyAcc (Not p) acc
   = "(~" ++ prettyAcc p (")"++acc)
prettyAcc (And p q) acc
   = "(" ++ prettyAcc p
              ("&&"++prettyAcc q (")"++acc))
prettyAcc (Imply p q) acc
   = "(" ++ prettyAcc p
              ("=>"++prettyAcc q (")"++acc))
```

Efficiency (cont.)

- We can improve readability by writing `prettyAcc` as a composition of functions of type `String -> String`
- This type is defined as `ShowS` in the Prelude

```
type ShowS = String → String -- in the Prelude
```

Efficiency (cont.)

```
prettyAcc :: Prop -> ShowS
prettyAcc (Const b)   = shows b
prettyAcc (Var x)     = (x:)
prettyAcc (Not p)
   = ("(~"++) . prettyAcc p . (")"++)
prettyAcc (And p q)   =
   = ("("++) . prettyAcc p . ("&&"++)
             . prettyAcc q . (")"++)
prettyAcc (Imply p q)
   = ("("++) . prettyAcc p . ("=>"++)
             . prettyAcc q . (")"++)
```