Homework 2*
SNU 4190.210, Fall 2011
Kwangkeun Yi

due: 9/30(Mon), 24:00

The objectives of this homework :

- Be familier with recursive program.
- Learn to make program using recursive data producing functions.
- Learn to make program with checking types.

Exercise 1 "Tree structure data"

Computer science always deals with trees, may it be a deep dark forest, large tree or spread branches.

A tree structure can be defined as :

- a basic tree structure: One leaf is a tree.
- a tree with subtrees: A point containing one or more subtrees is considered as a tree structure.

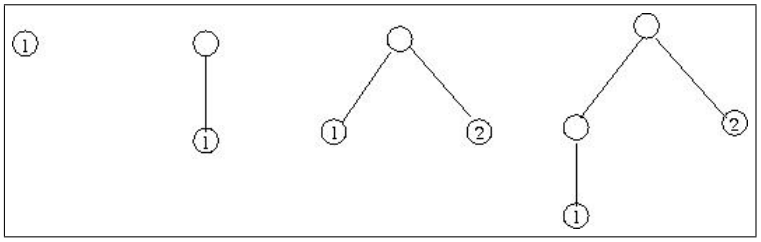
Above two conditions decide two methods of creating a tree structure. Making a basic tree (base case) and making a new tree using the existing ones (inductive case). Define the following two functions that build a tree structure:

`leaf` : $\alpha \rightarrow$ *tree*
`node` : *tree list* $\rightarrow$ *tree*

`leaf` makes a leaf tree with a arbitrary type (let's say "α"). In other words, `(leaf 1)` would make a leaf tree with an interger 1, `(leaf 'a)` would make a leaf tree with a symbol `a`, and `(leaf '(1 2))` would make a leaf tree with a list (1 2). `node` receives a list from the tree and makes a new tree that suspends the received lists in order. If `node` receives an emty list, it will create an empty tree.

For example, the below tree structures each

*translated by Youngseok Lee



shows a structures of a tree made from `(leaf '(1 2))`, `(node (list (leaf 1)))`, `(node (list (leaf 1) (leaf 2)))`, and `(node (list (node (list (leaf 1))) (leaf 2)))`.

Define the following four functions that use a tree:

`is-empty-tree?` : *tree* $\rightarrow$ *bool*
`is-leaf?` : *tree* $\rightarrow$ *bool*
`leaf-val` : *tree* $\rightarrow \alpha$
`nth-child` : *tree* $\times$ *nat* $\rightarrow$ *tree*

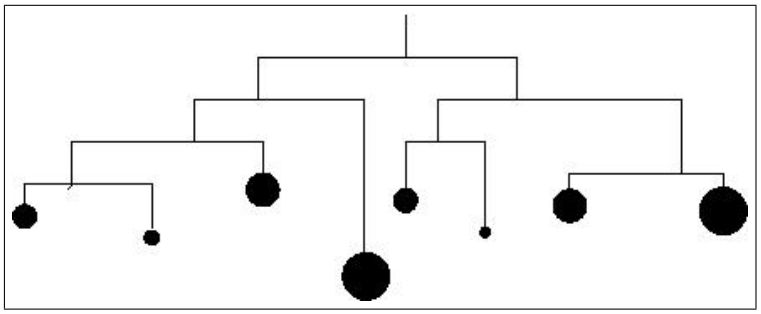
`is-empty-tree?` function determines whether a given tree is empty or not. `is-leaf?` determines whether a given tree is a leaf tree or not. `leaf-val` retrieves a value that a leaf has. `nth-child` receives a tree and a natural number $n \geq 0$ and returns a n-th subtree of the given tree. □

Exercise 2 "Measuring the weight of a mobile"

Let's think of a mobile that is hanging from the ceiling, balanced. A normal forked mobile would be defined as following:

- A single model is a mobile structure.
- Mobile with sub-mobiles: from a single balance point with sub-mobiles on the left/right sides hanging down would also be a mobile.

For example, the following diagram is one of the forked mobile:



Define the following three functions that makes a mobile structure. However, you must use the functions defined in Exercise 1:

`model` : $nat \to mobil$
`make-branch` : $nat \times mobil \to branch$
`make-mobil` : $branch \times branch \to mobil$

`model` receives a natural number and makes a mobile structure with that much weight. `make-branch` receives a length and a hanged mobile and creates one branch. `make-mobile` receives a branch from left/right and creates a single mobile.

Define the following two functions that uses mobile:

`is-balanced` : $mobil \to bool$
`weight` : $mobil \to nat$

`is-balanced?` determines whether a given mobile structure "is balanced". A single mobile is said "balanced" if the torque(length $\times$ weight) of each left and right branch of it is equal, and the total mobile's weight is the sum of the weight of the left and the right branch. If all the sub-mobiles of a single mobile is balanced, then it will be said to be balanced. `weight` gives out the total weight of a mobile. □

Exercise 3 "Logic circuit"

Boolean circuit can be defined inductively as followings. All the boolean circuit contains one or more input and a single output.

- base: A circuit with a value 0 is a boolean circuit.
- base: A circuit with a value 1 is a boolean circuit.
- induction: You can make a new circuit with a boolean circuit by using `not`
- induction: You can make a new circuit with a boolean circuit by using `and`
- induction: You can make a new circuit with a boolean circuit by using `or`

Define the above five methods to make a boolean circuit:

`zero` : *circuit*
`one` : *circuit*
`not-circuit` : $circuit \to circuit$
`and-circuit` : $circuit \times circuit \to circuit$
`or-circuit` : $circuit \times circuit \to circuit$

While defining the above methods, be sure to use the functions defined in Exercise 1 only. Also define the following six functions that use the boolean circuit:

`is-zero?` : $circuit \to bool$
`is-one?` : $circuit \to bool$
`is-not?` : $circuit \to bool$
`is-and?` : $circuit \to bool$
`is-or?` : $circuit \to bool$
`sub-circuit` : $circuit \times nat \to circuit$

`sub-circuit` receives a natural number $n \geq 0$ and returns the n-th sub-circuit. □

Exercise 4 "Calculation of the logic circuit"

Define the following function which calculates the final output value from the boolean circuit implemented in Exercise 3:

$$\texttt{output} : \textit{circuit} \rightarrow 0,1$$

The final outcome from the boolean circuit can be defined inductively as following. depending on how the circuit has been implemented. `zero` would output a 0. `one` would output a 1. `(not` B`)` would give 0 if the circuit B's output is 1, and vise-versa. `(and` B_1 B_2`)` would give out 1 only when both circuit B_1 and B_2 has an output of 1, and 0 otherwise. `(or` B_1 B_2`)` would give out 0 only when both circuit B_1 and B_2 has an output of 0, and 1 otherwise. □