CS 135
Winter 2014
Troy Vasiga

# Assignment: 6

**Due:** Tuesday, March 11, 2014 9:00pm
**Language level:** Beginning Student with List Abbreviations
**Allowed recursion:** Pure structural recursion
**Files to submit:** `bst.rkt`, `llt-height.rkt`, `siblings.rkt`, `beval.rkt`, and `familytree.rkt`
**Warmup exercises:** HtDP 12.2.1, 13.0.3, 13.0.4, 13.0.7, 13.0.8
**Practise exercises:** HtDP 12.4.1, 12.4.2, 13.0.5, 13.0.6

You can re-use the provided examples, but you should ensure you have an appropriate number of examples and a sufficient number of tests. All sorted lists are to be sorted in non-decreasing (ascending) order. Functions that produce ordered output should do so intrinsically, and must not sort or reverse an intermediate result. You may not use the Scheme functions *reverse* or *remove* or any functions not discussed in class. When in doubt, ask a question in Piazza. Here are the assignment questions you need to submit.

---

1. For this question, place your solution in the file `bst.rkt`.

   Recall from lectures, the definition of a binary search tree:

   ```
   (define-struct node (key val left right))
   ;; A binary search tree (Bst) is one of
   ;; * empty
   ;; * (make-node Num String Bst Bst),
   ;; where for each node (make-node k v l r), every node in the subtree
   ;; rooted at l has a key less than k, and every node in the subtree rooted
   ;; at r has a key greater than k.
   ```

   Sample binary search trees can be found in the provided `sample-bst.rkt` to test your functions. It is highly recommended that you also use your own sample binary search trees for test purposes.

   (a) Write the function *bst-add* that consumes a binary search tree *abst*, a key *k*, and a value *v*. (*bst-add abst k v*) produces the binary search tree resulting from adding the key *k*, with value *v* to *abst* (note that this addition will occur at the bottom of the tree). If *abst* already contains the key *k* it should be updated so that it associates the key *k* with the value *v*, instead of whatever value used to be associated with *k*.

   (b) A binary tree is called *full* if every node possesses either 0 or 2 children. Write the predicate function *bst-full?* that consumes a binary search tree and produces *true* if the given binary search tree is full, and *false* otherwise. An empty tree is considered full.

(c) Write a function *bst-height* that consumes a binary search tree and produces the height of the tree. The height of a binary tree is the maximum distance from the root to a leaf, measured in nodes (including the root and the leaf). In particular, the empty tree has height 0, and a single node has height 1. For example, the binary tree

```
(make-node 2 "" (make-node 1 "" empty empty)
                (make-node 3 "" empty (make-node 4 "" empty empty)))
```

has height 3, since a path from the root (2) to a leaf (4) encounters three nodes along the way.

(d) A binary tree is called *perfect* if it is full, and additionally if every leaf in the tree has the same *depth* (i.e., distance from the root). Equivalently, a tree is perfect if every node's left subtree and right subtree have the same height. Write the predicate function *bst-perfect?* that consumes a binary search tree and produces *true* if the given binary search tree is perfect, and false otherwise. The empty tree is perfect. **Hint**: Use the function *bst-height* as a helper function.

(e) Write a function *bst->sal* that consumes a binary search tree and produces an association list (*AL*) that contains all entries in the tree, sorted by their keys in ascending order. **Note: for full marks, you cannot compute an intermediate, unsorted *AL* and then sort it afterwards.** Note that you are allowed to use the function *append* for this part.

2. For this question, place your solution in the file `llt-height.rkt`.

Recall from lectures, the definition of a leaf-labelled tree:

```
;; A leaf-labelled tree (Llt) is one of the following:
;; * empty
;; * (cons l1 l2), where l1 is a non-empty Llt and l2 is an Llt
;; * (cons v l), where v is an Int and l is an Llt
```

Write the Scheme function *height* which consumes a leaf-labelled tree *t* and produces the height of *t*. The height of a leaf-labelled tree is:

- 0, if the tree is empty;
- 1, if the tree contains only one leaf node;
- 1+(maximum of all heights of the subtrees) in all other cases.

For example,

```
(height empty) ⇒ 0
(height (list 1 2 3)) ⇒ 1
(height (list (list 1) (list (list 8 9)))) ⇒ 3
```

3. For this question, place your solution in the file `siblings.rkt`.

   Using the same definition for a leaf-labelled tree as in the write the Scheme function *num-siblings* which consumes a leaf-labelled tree *t* (which has **distinct** labels), and an integer *n*. The function should produce the number of leaves that are siblings of *n* in *t* (if *n* is a leaf in *t*), otherwise, the function should produce *false* (if *n* is not a leaf in *t*). Two leaf nodes are siblings if they have the same parent in the tree. For example

   (*num-siblings* (*list* (*list* 1 2) 3 (*list* 4)) 3) ⇒ 0
   (*num-siblings* (*list* (*list* 1 2 3) (*list* (*list* 4 5 6))) 5) ⇒ 2
   (*num-siblings* (*list* (*list* 1 2 3) (*list* (*list* 4 5 6))) 9) ⇒ *false*

   Note: you may use the function member? in this question.

4. For this question, place your solution in the file `beval.rkt`, which has been created with some structure and definitions for you already.

   (a) Create one template for each of *Bexp*, *Cexp*, *Cbexp* and *Cbexplist*. Do this part before part (b). Yes, really.

   (b) Using your templates, write the Scheme function *bool-eval* which consumes a boolean expression *bexpr* and produces the boolean value of the boolean expression based on the evaluation rules in Module 3. For example:

   (*bool-eval* (*make-comp-exp* '< 4 7)) ⇒ *true*
   (*bool-eval* (*make-compoundb-exp* 'or *empty*) ⇒ *false*
   (*bool-eval* (*make-compoundb-exp*
   'and
   (*list* (*make-comp-exp* '> 5 2) *true true*))) ⇒ *true*

This concludes the list of questions for which you need to submit solutions.

---

5. **5% Bonus**: Consider the following structure and data definitions:

```
(define-struct node (name left right))
;; A binary descendants' tree (BDTree) is one of
;; * empty
;; * a (make-node String BDTree BDTree)
```

A BDTree can be used to model genealogical data, under the simplifying assumption that every person named in the tree has at most two children. For example, if Alice has children Bob and Charlie, Bob has children Daisy and Edward, and Charlie had children Fred and Gertrude, then the following BDTree models this information:

```
(define family (make-node "Alice"
                          (make-node "Bob" (make-node "Daisy" empty empty)
                                           (make-node "Edward" empty empty))
                          (make-node "Charlie" (make-node "Fred" empty empty)
                                               (make-node "Gertrude" empty empty))))
```

Two family members are considered siblings if they have the same parent, first cousins if they have different parents, but the same grandparent, second cousins if they have the same grandparent, but different great grandparents, etc. For the sake of uniformity, we can consider siblings to be 0th cousins. For example, in the tree above, Bob and Charlie are 0th cousins, as are Daisy and Edward, and Fred and Gertrude. However, Daisy and Fred are first cousins, as are Daisy and Gertrude, Edward and Fred, and Edward and Gertrude. Children of Daisy would be first cousins with children of Edward, but second cousins with children of Fred, etc.

Put another way, two family members are $m$-th cousins if their closest common ancestor is $m+1$ generations above them.

More generally, two family members are $m$-th cousins, $n$ times removed if their closest common ancestor is $m+1$ generations above one of them, and $m+n+1$ generations above the other. For example, Daisy would be a first cousin, once removed with children of Fred or Gertrude. Although Charlie would be considered Daisy's uncle, we could also say that Charlie and Daisy are 0th cousins, once removed. Daisy's children would be Charlie's 0th cousins, twice removed.

Write a Scheme function called *cousins?* that consumes a BDTree, two names, and natural numbers *m* and *n*. The expression (*cousins? abdtree name1 name2 m n*) produces true if *name1* and *name2* are $m$-th cousins, $n$ times removed in *abdtree*. You may assume that both given names appear in the tree, that no name appears more than once in the tree. Direct descendants are not cousins of any kind, so for example, in the given tree, Alice is not any kind of a cousin with any other member of the tree.

Place your solution in the file `familytree.rkt`.

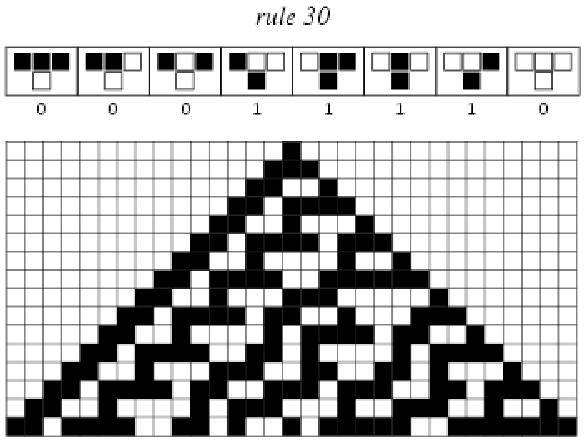
---

**Enhancements**: *Reminder—enhancements are for your interest and are not to be handed in.*

A cellular automaton is a way of describing the evolution of a system of cells (each of which can be in one of a small number of states). This line of research goes back to John von Neumann, a mathematician who had a considerable influence on computer science just after the Second World War. Stephen Wolfram, the inventor of the Mathematica math software system, has a nice way of describing simple cellular automata. Wolfram believes that more complex cellular automata can serve as a basis for a new theory of real-world physics (as described in his book "A New Kind of Science", which is available online). But you don't have to accept that rather controversial proposition to have fun with the simpler type of automata.

The cells in Wolfram's automata are in a one-dimensional line. Each cell is in one of two states: white or black. You can think of the evolution of the system as taking place at clock ticks. At one tick, each cell simultaneously changes state depending on its state and those of its neighbours to the left and right. Thus the next state of a cell is a function of the current state of three cells. There are thus 8 ($2^3$) possibilities for the input to this function, and each input produces one of two values; thus there are $2^8$ or 256 different automata possible.

If white is represented by 0, and black by 1, then each automaton can be represented by an 8-bit binary number, or an integer between 0 and 255. Wolfram calls these "rules". Rule 0, for instance, states that no matter what the states of the three cells are, the next state of the middle cell is white, or 0. But Rule 1 says that in the case where all three cells are white (the input is 000, which is zero in binary), the next state of the middle cell is black (because the zeroth digit of 1, or the digit corresponding to the number of $2^0$s in 1, is 1, meaning black). In the other seven cases, the next state is white.



This is all made clearer by the pictures at the following URL, from which the picture at the right is taken:

`http://mathworld.wolfram.com/CellularAutomaton.html`

Some of these rules, such as rule 30, generate unpredictable and apparently chaotic behaviour (starting with as little as one black cell with an infinite number of white cells to left and right); in fact, this is used as a random number generator in Mathematica.

You can use DrScheme to investigate cellular automata and draw or animate their evolution, using the `io.ss`, `draw.ss`, `image.ss`, or `world.ss` teachpacks. Write a function that takes a rule number and a configuration of cells (a fixed-length list of states, of a size suitable for display) and computes the next configuration.