Lab 11: General trees

Create a separate file for each question. Keep them in your "Labs" folder, with the name `liqj` for Lab i, Question j.

This lab makes use of the following structure and data definitions:

(*define-struct t-node* (*label children*))
;; A general tree (**gen-tree**) is either
;; a string or
;; a structure (*make-t-node l c*), where
;; *l* is a string and
;; *c* is a *tree-list*.

;; A **tree-list** is either
;; (*cons t empty*), where
;; *t* is a *gen-tree* or
;; (*cons t tlist*), where
;; *t* is a *gen-tree* and
;; *tlist* is a *tree-list*.

(*define-struct single-product* (*name origin*))
;; A **single-product** is a structure (*make-single-product n o*), where
;; *n* is a string and
;; *o* is a string denoting the country of origin.

(*define-struct sales-product* (*ID prod*))
;; A **sales-product** is a structure (*make-sales-product i p*), where
;; *i* is an integer and
;; *p* is either a single-product or a product-list.

;; A **product-list** is either
;; *empty* or
;; (cons sp pl), where
;; *sp* is a sales-product and
;; *pl* is a product-list.

Download the headers for each function from the file `labinterface11.rkt` linked off the "Labs" page on the course Web site.

After you have completed a question (except class exercises), including creating tests for it, you can obtain feedback by submitting it and requesting a public test. Follow the instructions given in the Style Guide.

The teachpack `compound.rkt` contains structures for manipulation of chemical compounds (*compound*s, *part*s, and *element*s); details can be found in `compound.pdf` on the course Web site. Remember to add the teachpack.

Language level: Beginning Student with List Abbreviations

1. *[Class exercise with lab instructor assistance]* Create a function *node-count* that consumes a *gen-tree g* and produces the number of nodes (both leaves and internal nodes).

2. We can modify an arithmetic expression to include a variable x, denoted 'x, as one of the base cases. Write a data definition for arithmetic expressions with variables (*aevx*), and then create functions *evalx* and *applyx* needed to evaluate an expression, given a value for 'x.

3. Create a function *llt-count* that consumes a value and a *llt* and counts the number of times that value appears in the *llt*. Do not use flatten.

4. Create the function *from-country?* that consumes a *sales-product* and a *string* indicating country of origin and produces *true* if there is a *single-product* within the *sales-product* with that origin, and *false* otherwise. For example, using constants declared in the *product* teachpack:

   - (*from-country? lipbalmpack* "Denmark") => *true*
   - (*from-country? lipkit* "Denmark") => *true*
   - (*from country? promokit* "Kenya") => *false*

   As this is a new type for you, be sure to develop templates for functions involving *single-product*, *sales-product*, and *product-list* before attempting to write this function.

5. Create a function *molar-mass* that consumes a *compound* and produces the total molar mass of the *compound*. Note the following when calculting the molar mass:

   - The molar mass of an *element* is given by its field *mmass*.
   - The molar mass of a *compound* is the sum of the molar masses of each of its *part*s.
   - The molar mass of a single *part* is the product if its *size* and the molar mass of its *eoc* field.

   For example, the molar mass of *c489* is 489. Remember to use *check-within* for testing if the data you are using produces inexct numbers.

6. *Optional open-ended questions* A program that evaluates an arithmetic expression is the starting point of a Scheme interpreter. Using a small subset of Scheme, write an interpreter for a Scheme expression. Once you have it working, consider adding other aspects of Scheme to enhance your interpreter.