CS 135 Fall 2017
Becker, Goldberg, Kaplan, Nijjar, Tompkins, Vasiga

Assignment: 7

Due: Tuesday, November 14th, 2017 9:00pm
Language level: Intermediate Student
Allowed recursion: Pure Structural and Structural Recursion with an Accumulator
Files to submit: `trie.rkt`, `moretabular.rkt`, `bonus.rkt`
Warmup exercises: HtDP 14.2.1, 14.2.2, 15.1.1, 15.1.3, 18.1.1, 18.1.2, 18.1.3, 18.1.4, 18.1.5
Practice exercises: HtDP 14.2.3, 14.2.4, 14.2.6, 15.1.2, 15.1.4, 18.1.5

- **Make sure you read the OFFICIAL A07 post on Piazza** for the answers to frequently asked questions.
- Unless stated otherwise, all policies from Assignment 06 carry forward.
- You may use the following list functions and constants: *cons*, *cons?*, *empty*, *empty?*, *first*, *second*, *third*, *rest*, *list*, *append*, *length*, *string->list*, *list->string*. You may **not** use *reverse* (or define your own version) unless specified in the question.

注意，只有若干题允许用 reverse，读题！

- Some correctness marks will be allocated for avoiding exponential blowups, as demonstrated by the *max-list* example in slide 07-6. (But note that not every repeated function invocation leads to an exponential blowup.)

真的按照套路写，都出现这个就见鬼了

- You may not use abstract list functions.

Caution: Sketch out solutions to these problems on paper before implementing them. If you try to compose this assignment in DrRacket directly without any pre-planning, you will with high probability end up frustrated.

Here are the assignment questions you need to submit.

1. Perform the assignment 7 questions using the online evaluation "Stepping Problems" tool linked to the course web page and available at

   `https://www.student.cs.uwaterloo.ca/~cs135/stepping`.

   The instructions are the same as those in assignment 3; check there for more information if necessary.

   The use of **local** will involve defining constants and moving them to global scope. Don't get stuck because you are trying to re-submit those constants once they are in simplest form.

   已经 simplify 的 constant，不要再 submit 一次

一个专门用来找单词的 tree，我们用来做自动补全，

2. A **trie** (usually pronounced "try") is a type of search tree suitable for searching through collections of words. In the real world, trie-like structures can be useful for autocompletion and text compression.

   Your tries will be made up of TNodes, which are defined using a structure:

```
(define-struct tnode (key ends-word? children))
;; A TNode is a (make-tnode Char Bool (listof TNode))
;; requires: the TNodes of children are sorted in lexicographical
;; order by key.
```

   每一个 tnode 包含一个字母（当前字母）
   一个 Bool：true 如果这个位置结束可以代表一个单词
   a listof tnode：children（可以为空）
   children 都是用它们的字母排序的（char<?）

   We can then define a Trie:

```
(define-struct trie (children))
;; A Trie is a (make-trie (listof TNode))
;; requires: the TNodes of children are sorted in lexicographical
;; order by key.
```

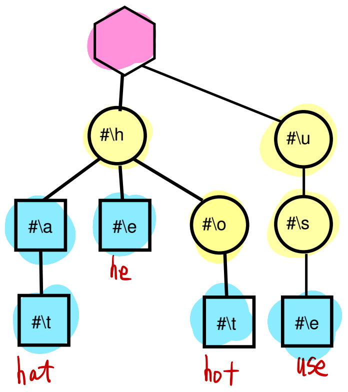
   Each key of a TNode is a character. Each TNode represents a character of at least one word in the trie.

   TNodes can have any number of children, but ordering is important: the children are sorted by key in lexicographic order as determined by the *char<?* function (note that this *is* case sensitive.) No two children of a TNode will have the same *key* value.

   大小写

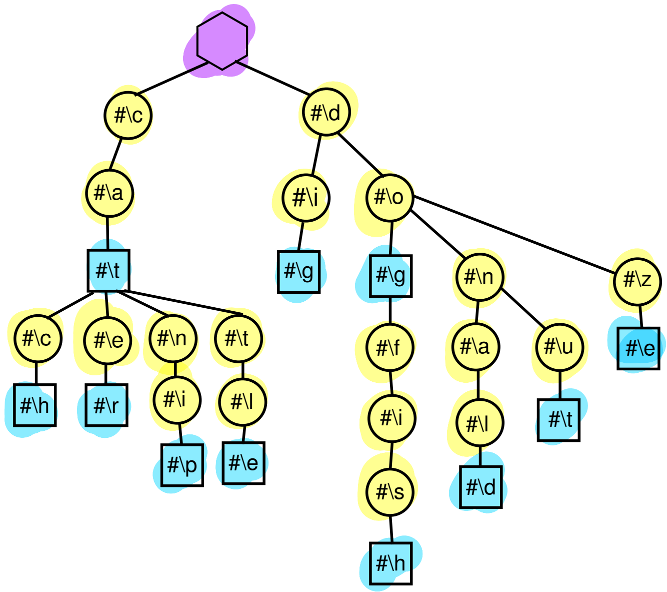
   In our implementation, the empty string **""** is not a legitimate word, and cannot be represented as a word in our Trie.

   Here is a visual depiction of a Trie called *h-u-trie*. In the diagram, circular nodes represent TNodes that do not end words (that is, *ends-word?* is false), square nodes represent TNodes that do end words, and the hexagon is the Trie structure. This trie represents the words "hot", "hat", "ha", "he" and "use". Note that "ho" and "us" are not words in this trie even though according to Santa Claus they are words in English.



○代表的都不能成为一个单词，h，ho，us
□代表的都可以，ha hat he，hot，use
↓
因为 ○ 部分的 bool 是 false □ 的地方是 true

   Here is a more complicated trie, called *c-d-trie*:



The above trie represents the words “cat”, “catnip”, “dog” “donut”, “doze”, “catch”, “cattle”, “cater”, “donald”, “dogfish”, and “dig”.

These tries (and a few more, as well as structure/data definitions) are contained in *a07lib.rkt* . You may find them helpful for testing your own code. Similar to the helper files for assignments 4 and 5, you can include this file in your own code by including the following line the near top of *trie.rkt*:

(*require* "a07lib.rkt")

In this question you will write functions to create and query tries. Even though many of the functions below consume or produce strings, the **only** string functions you may use in this question are *string->list* and *list->string*. This means that many of these functions will be wrapper functions. The recursive helper functions you write will operate on lists of characters, similar to the example given in slides 05-46 to 05-49, or Question 2 of Assignment 6.

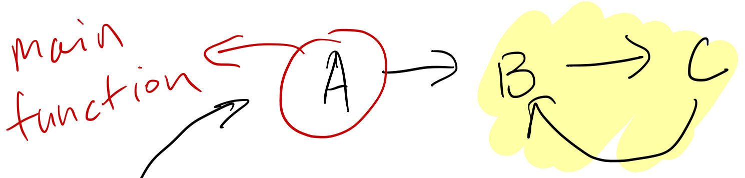
不可以给
string 直接
recursion, 必须
用 listof chars

This restriction on string functions also means you should not determine a string’s length with *string-length*, or sort a list of strings (because that uses *string<?*).

For this question you are not required to use **local** to encapsulate helper functions or to give names to values you would like to reuse (but you may use **local** if you wish). You may find that helper functions in one part of the question will be helpful in completing other parts.

Place your solutions in the file `trie.rkt`

(a) Write a constant *a-tnode* which uses TNodes to define the word “at”. 先画图再做!
Write a constant *c-tnode* which uses TNodes to define the following words: “cs135”, “cow”, “cs136”, “cs115”, “cs116”, “coo”. (Note that due to DrRacket’s indentation rules your constant definition might exceed 80 characters in width. This is acceptable for this constant.)



B, C是 mutual Recursion, 因为 B处理的是 listof tnode, C处理的是一个tnode, B要用C处理 list里面的每一个node, 而C又要用B处理 children

Write a constant *a-c-trie* which defines a Trie containing *a-tnode* and *c-tnode*:

(*make-trie* (*list a-tnode c-tnode*))

You can use these constants to help you in developing and testing the rest of your functions. We will include all correctness tests for this part in the basic tests, so that you can be sure that you understand the underlying trie structure. (This also means that passing the basic tests for this part means you get full correctness marks for it.)

(b) Write three template functions that together operate on Tries. Many (but not necessarily all) of the subsequent functions you write will be based on these templates. 让你写个套路出来

- A A function *trie-template* which consumes a Trie and produces Any. This function should call *list-tnode-template*. structure的套路
- B A function *list-tnode-template* which consumes a list of TNodes and produces Any. This function will be mutually recursive on *tnode-template*. list的套路
- C A function *tnode-template* which consumes a TNode and produces Any. This function will be mutually recursive on *list-tnode-template*.

Comment out these template functions in your code, but include them with your submission.

(c) Write a function *in-trie?* which consumes a string and a Trie, and produces *true* if the string is represented as a word in the trie. For example, (*in-trie?* "cs" *a-c-trie*) produces *false*, (*in-trie?* "cs115" *a-c-trie*) produces *true* and (*in-trie?* "cower" *a-c-trie*) produces *false*. 看看一个单词在不在一个trie里（你们做过一个很接近的问题在A6）

(d) Write a function *list-words* which consumes a Trie and produces a list of all the words in the Trie. For full marks your recursion should produce the list of words in sorted order. Producing a correct list of unsorted words will be worth fewer marks, but still be helpful in testing functions below. Sorting the list of words, removing duplicates, or otherwise manipulating the produced list of words after it has been generated by traversing the Trie will be worth no more marks than producing an unsorted list of words.

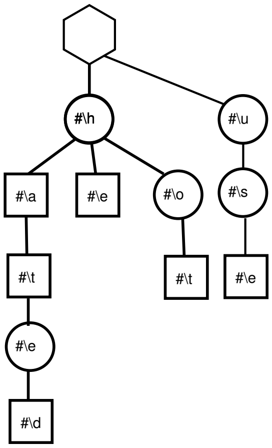
For example, (*list-words h-u-trie*) produces (*list* "ha" "hat" "he" "hot" "use").

For this question you may use the Racket *reverse* function, but only to reverse a list of characters. You will find at least one accumulator helpful.

把一个trie能表达的所有单词用从小到大的字母顺序 produce出来，你肯定不须要另外的排序，只要你按照list本来的顺序就可以了，别忘了，children已经是按照char的大小排序了

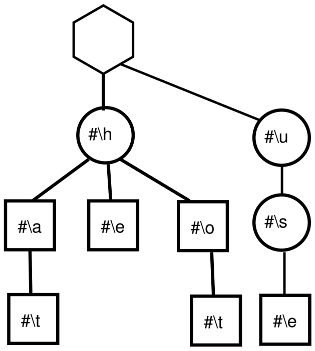
(e) Write a function *insert-word* which consumes a string and a Trie, and produces a Trie consisting of the consumed Trie with the word inserted. The word may already be represented in the trie, but it is acceptable for your code to go through the motions of inserting it again.

这个作业比较有难度的一个问题，提示，处理把一个tnode insert进children的时候，还是很像排序一个list的insert，都是要找到正确的位置



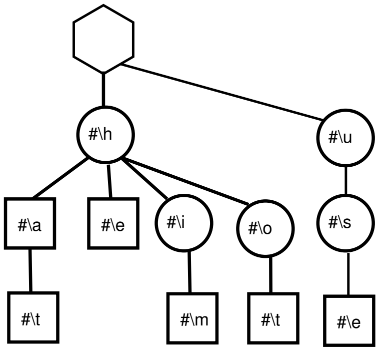
For example, (*insert-word* "hated" *h-u-trie*) produces this trie, and (*list-words* (*insert-word* "hated" *h-u-trie*)) produces (*list* "ha" "hat" "hated" "he" "hot" "use").

Here is a second example:



(*insert-word* "ho" *h-u-trie*) produces this trie, and (*list-words* (*insert-word* "ho" *h-u-trie*)) produces (*list* "ha" "hat" "he" "ho" "hot" "use").

And here is a third:



(*insert-word* "him" *h-u-trie*) produces this trie, and (*list-words* (*insert-word* "him" *h-u-trie*)) produces (*list* "ha" "hat" "he" "him" "hot" "use").

直接看insert-word的结果不好测试，用 list-words 比较方便，但是同时

Hint: Using *list-words* in your tests is valid and considerably easier than hand-coding the expected trie. However, when we test correctness we will be looking at the structure of your tries and not just the output of *list-words*, so you may need some additional tests to ensure your tries do not have structural problems. 你需要其它helper帮助你检查是否按顺序，还有是否有重复

(f) Write a function *insert-some-words* which consumes a list of strings and a Trie, and produces a Trie consisting of the consumed Trie with all of the strings inserted. For example, (*list-words* (*insert-some-words* (*list* "hog" "hoot") *h-u-trie*)) produces (*list* "ha" "hat" "he" "hog" "hoot" "hot" "use"). 放进去一堆，你就是不会写前面任何一道题这个都应该能做，跟trie没关系了，说白了就是sort的代码

(g) Write a function *list-completions* which consumes a string and a Trie, and produces a sorted list of strings consisting of all words in the Trie that begin with that prefix.

(d)问的加强版，先找，再返回所有，d问会是你的好帮手

For example, (*list-completions* "don" *c-d-trie*) produces (*list* "donald" "donut"), (*list-completions* "ha" *h-u-trie*) produces (*list* "ha" "hat"), (*list-completions* "" *h-u-trie*) produces (*list* "ha" "hat" "he" "hot" "use"), and (*list-completions* "go" *h-u-trie*) produces *empty*.

Again, you may use *reverse* in this function, but only to reverse a list of characters.

3. Recall the definition of Table from assignment 6.

In this question you will write additional functions that work with Tables, with an additional restriction: *all* the helper functions you write must be encapsulated using **local**. You may, however, define separate constants for your tests.

Place your solutions in `moretabular.rkt`.

(a) Write a function *mirror* which consumes a table and reverses the elements of each row. You may not use *reverse* for this function but you are allowed to implement your own version as an encapsulated helper function.

map问题!

accumulative Recursion的特点之一就是在递的时候进行组合，所以呢？什么顺序？

For example, (*mirror* '((-3.2 4.5 7) (13 3 -3))) produces (*list* (*list* 7 4.5 −3.2) (*list* −3 3 13)).

(b) Write a function *element-apply-many* which consumes a list of functions (each with contract *Num* → *Num*, *Num* → *Int*, or *Num* → *Nat*) and a table, and produces a list of tables. The first table should be the results of applying the first function to each table element, the second table should be the results of applying the second function to each table element, and so on. For example:

还是map 问题!

(**define** (*add3 x*) (+ *x* 3))
(*element-apply-many* (*list abs floor add3*)
'((7 4.5 -3.2)(-3 3 13)))

produces
(*list* (*list* (*list* 7 4.5 3.2)
(*list* 3 3 13))
(*list* (*list* 7 4 −4)
(*list* −3 3 13))
(*list* (*list* 10 7.5 −0.2)
(*list* 0 6 16)))

每个做abs

每个做floor

每个做add3

假设我们有一个 listof functions
(define L (list abs floor add3))
那么
((first L) -5) ⇒ 5
((third L) 3) ⇒ 6

(c) Write a function *scale-smallest* which consumes a non-empty table (with at least one column and one row) and a real number (the *offset*). This function produces a second function that consumes a number, multiplies that number by the smallest element of the table, and adds the offset.

produce 你的 local function!

To test this function, you may define an additional helper function in the global scope as follows:

;; (apply-function f arg) produces the result of f with the given argument arg.
;; apply-function: (X → Y) X → Y
(**define** (*apply-function f arg*)
(*f arg*))

scale-smallest 拿到一个 table (listof (listof Num)) 和一个数字，产生一个function (产生的这个function 再 consume 一个数字) 在这个产生的 function 里面会把本来 table 里面最小的数字乘以新产生的 function consume的数字再加上 scale-smallest 本来 consume 的数字

把下一页的例子一定要看懂.

Then (*apply-function* (*scale-smallest* '((7 4.5 3.2)(-3 3 13)) 2.4) 7) produces -18.6 because $-3 \cdot 7 + 2.4 = -18.6$. Similarly, (*apply-function* (*scale-smallest* '((7 4.5 3.2)(-3 3 13)) 2.4) -2.7) produces 10.5.

Be careful to avoid exponential blowups in your function implementation.

This concludes the list of questions for which you need to submit solutions. As always, do not forget to check your email for the basic test results after making a submission.

Bonus: For a 5% bonus, write a function *remove-word*, which consumes a string and a Trie, and produces a Trie with the string removed if it is represented in the Trie. The produced Trie should not contain any unnecessary TNodes. (That is, every leaf TNode in the Trie should end a word.) If the string is not represented in the Trie then the function should produce the original Trie.

You may not use *insert-word*, *insert-some-words* or analogous functions you rewrite in your solution.

For example, (*remove-word* "hated" (*insert-word* "hated" *h-u-trie*)) should produce a Trie identical to *h-u-trie*.

Place your solutions in `bonus.rkt` . If you require functions from `trie.rkt` in your solution, paste them into `bonus.rkt`.

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

The material below first explores the implications of the fact that Racket programs can be viewed as Racket data, before reaching back seventy years to work which is at the root of both the Scheme language and of computer science itself.

The text introduces structures as a gentle way to talk about aggregated data, but anything that can be done with structures can also be done with lists. Section 14.4 of HtDP introduces a representation of Scheme expressions using structures, so that the expression (+ (∗ 3 3) (∗ 4 4)) is represented as

```
(make-add
   (make-mul 3 3)
   (make-mul 4 4))
```

But, as discussed in lecture, we can just represent it as the hierarchical list '(+ (∗ 3 3) (∗ 4 4)). Scheme even provides a built-in function *eval* which will interpret such a list as a Scheme expression and evaluate it. Thus a Scheme program can construct another Scheme program on the fly, and run it. This is a very powerful (and consequently somewhat dangerous) technique.

Sections 14.4 and 17.7 of HtDP give a bit of a hint as to how *eval* might work, but the development is more awkward because nested structures are not as flexible as hierarchical lists. Here we will use the list representation of Scheme expressions instead. In lecture, we saw how to implement *eval* for expression trees, which only contain operators such as +,−,∗,/, and do not use constants.

Continuing along this line of development, we consider the process of substituting a value for a constant in an expression. For instance, we might substitute the value 3 for *x* in the expression $(+\ (*\ x\ x)\ (*\ y\ y))$ and get the expression $(+\ (*\ 3\ 3)\ (*\ y\ y))$. Write the function *subst* which consumes a symbol (representing a constant), a number (representing its value), and the list representation of a Scheme expression. It should produce the resulting expression.

Our next step is to handle function definitions. A function definition can also be represented as a hierarchical list, since it is just a Scheme expression. Write the function *interpret-with-one-def* which consumes the list representation of an argument (a Scheme expression) and the list representation of a function definition. It evaluates the argument, substitutes the value for the function parameter in the function's body, and then evaluates the resulting expression using recursion. This last step is necessary because the function being interpreted may itself be recursive.

The next step would be to extend what you have done to the case of multiple function definitions and functions with multiple parameters. You can take this as far as you want; if you follow this path beyond what we've suggested, you'll end up writing a complete interpreter for Scheme (what you've learned of it so far, that is) in Scheme. This is treated at length in Section 4 of the classic textbook "Structure and Interpretation of Computer Programs", which you can read on the Web in its entirety at `http://mitpress.mit.edu/sicp/` . So we'll stop making suggestions in this direction and turn to something completely different, namely one of the greatest ideas of computer science.

Consider the following function definition, which doesn't correspond to any of our design recipes, but is nonetheless syntactically valid:

```
(define (eternity x)
  (eternity x))
```

Think about what happens when we try to evaluate (*eternity* 1) according to the semantics we learned for Scheme. The evaluation never terminates. If an evaluation does eventually stop (as is the case for every other evaluation you will see in this course), we say that it *halts*.

The non-halting evaluation above can easily be detected, as there is no base case in the body of the function *eternity*. Sometimes non-halting evaluations are more subtle. We'd like to be able to write a function *halting?*, which consumes the list representation of the definition of a function with one parameter, and something meant to be an argument for that function. It produces *true* if and only if the evaluation of that function with that argument halts. Of course, we want an application of *halting?* itself to always halt, for any arguments it is provided.

This doesn't look easy, but in fact it is provably impossible. Suppose someone provided us with code for *halting?*. Consider the following function of one argument:

```
(define (diagonal x)
  (cond
    [(halting? x x) (eternity 1)]
    [else           true]))
```

What happens when we evaluate an application of *diagonal* to a list representation of its own definition? Show that if this evaluation halts, then we can show that *halting?* does not work correctly for all arguments. Show that if this evaluation does not halt, we can draw the same conclusion. As a result, there is no way to write correct code for *halting?*.

This is the celebrated *halting problem*, which is often cited as the first function proved (by Alan Turing in 1936) to be mathematically definable but uncomputable. However, while this is the simplest and most influential proof of this type, and a major result in computer science, Turing learned after discovering it that a few months earlier someone else had shown another function to be uncomputable. That someone was Alonzo Church, about whom we'll hear more shortly.

For a real challenge, definitively answer the question posed at the end of Exercise 20.1.3 of the text, with the interpretation that *function=?* consumes two lists representing the code for the two functions. This is the situation Church considered in his proof.