Assignment: 7
Due: Tuesday, March 15th, 2022 9:00pm
Language level: Beginning Student with list abbreviations
Allowed recursion: Simple recursion and Accumulative recursion
Files to submit: bst.rkt, trie.rkt, eval-apply-a.rkt, eval-apply-b.rkt, eval-apply-c.rkt, bonus.rkt, acadinteg-a07.txt
Warmup exercises: 14.2.1, 14.2.2, 15.1.1, 15.1.3
Practice exercises: 14.2.3, 14.2.4, 14.2.6, 15.1.2, 15.1.4

• 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.
• This assignment covers material up to Slide 17 of Module 13.
• Remember that basic tests are meant as sanity checks only; by design, passing them should not be taken as any indication that your code is correct, only that it has the right form.
• Some correctness marks will be allocated for avoiding exponential blowups, as demonstrated by the max-list example in slide 09-4. (But note that not every repeated function invocation leads to an exponential blowup.)
• Download the file acadinteg-a07.txt from the assignment page. Fill in your Quest userid in the provided place to indicate that you have read and understood the policy. We take this to be equivalent to signing the document, so save a copy for your own records.
• Solutions will be marked for both correctness [80%] and style [20%]. Follow the guidelines in the Style Guide.
• You may not use abstract list functions.

Here are the assignment questions you need to submit.

1. [10+10= 20% Correctness] Binary search trees (BSTs) are a commonly used structure to store data in an ordered fashion. In the context of this question, imagine a BST populated with Nats.

   ;; A Binary Search Tree (BST) is one of:
   ;;  * empty
   ;;  * a Node
(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; requires: key > every key in left BST
;; key < every key in right BST

An example of a BST would be:

(define my-bst (make-node 5 (make-node 3 empty empty) (make-node 9
(make-node 7 empty empty) empty)))

(a) Write a function bst? that consumes an Any and produces true if it is a BST and false otherwise.

(check-expect (bst? 5) false)
(check-expect (bst? (make-node false false false)) false)
(check-expect (bst? my-bst) true)

Hint: Some approaches to this problem are simpler if you know that +inf.0 and -inf.0 are special numbers in Racket meaning positive infinity and negative infinity, respectively.

(b) Write a function bst-add that consumes a key (as Nat) and a BST and produces a new BST with a new node added, where the new node contains the key.

Example:

(check-expect (bst-add 4 my-bst) (make-node 5 (make-node 3 empty (make-node 4 empty empty))
(make-node 9 (make-node 7 empty empty) empty)))

Place your solutions to this problem in bst.rkt.

2. [4+4+8= 16% Correctness]

(a) Implement the alternate arithmetic expression trees discussed briefly on slide 17 in Module 13 (check the slide!). Place your solutions to this problem in eval-apply-a.rkt.

(b) Modify the data definition so that an expression can contain constants. Modify eval and apply to also consume a dictionary of constants, represented by an association list, and their values.

Examples:

(define constants '((x 10) (y 20)))
(check-expect (eval '(+ x y 10) constants) 40)
Place your solutions to this problem in `eval-apply-b.rkt`.

(c) Minimally change the data definition to enhance it so that it includes Boolean values `true` and `false` and the Boolean operations `and`, `or`, `=`, and `not`. Don't forget to include the requirements for `not`. Enhance your functions to implement the new data definition. Make sure your code runs on the basic tests. (Note that `and`, `or`, and `=` should accept any arbitrary number of arguments.)

Examples:

```racket
(define constants '((x 10) (y 20)))
(check-expect (eval (list 'and (list '= 'x 10) (list '= 'y 0) true) constants) false)
```

The second test can be written as

```racket
(check-expect (eval '((and (= x 10) (= y 0) #true) constants) false)
```

Note that we need to use `#true` and `#false` instead of `true` and `false` if we are using the quoting for list notation.

Place your solutions to this problem in `eval-apply-c.rkt`.

3. `[4+6+6+(5)+10+6+6+6= 44% + (5% bonus) Correctness]

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:

```racket
(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.
```

We can then define a Trie:

```racket
(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.
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.

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 assignment 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.

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<?).

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”.

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

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.

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

Do not comment out your template functions in your code.

(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.

(d) Bonus– There are more non-bonus (required) questions below. for 5% Write a function list-nodes which consumes a Trie and produces the list of the keys (the
character part) of all the nodes in the Trie. Please note that all there might be repetition in the list. The order of the appearance of keys in the list should follow the rules below:

- If two keys are from the nodes with different depths, the key from the node with the smaller depth should appear before the other node.
- If two keys are from the nodes with the same depth, the smaller key should appear before the other node.

(e) 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. Please see this Piazza post, for the changes regarding part (d) and some hints for this part.

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.

(f) 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.

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:

```racket
#<trie>
```

CS 135 — Winter 2022  
Assignment 7  
6
(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").

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.

(g) 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").

(h) 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. For example, (list-completions "don" c-d-trie) produces (list "donald" "donut"), (list-completions "ha" h-u-trie) produces (list "ha" "hat" "he" "hoc" "hoot" "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.

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.
4. **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]))
```

CS 135 — Winter 2022 Assignment 7
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.