Scheming with Scheme and Grammar

10/9: Corrected an error related to incorrect ordering on the simplify examples

Graphs

Graphs

For this problem, we represent an unweighted directed graph as a list of edges where an edge is a list of length 2 containing a start-node and an end-node. Nodes are always given as symbols. Write the following graph-related functions.

Functions

1. `(graph-create)` Returns a graph with no edges.
2. `(graph-remove-node g n)` Returns the graph that results from removing node n from graph g.
3. `(graph-add-edge g e)` Returns the graph that results from adding edge e to graph g. The edge is a list of length two.
4. `(graph-add-edges g froms tos)` Returns the graph that results from adding the edges expressed as two lists, froms and tos, to graph g. List `froms` is of the same length as list `tos` and each list contains only nodes.
5. `(graph-shortest-path g s t)` Returns a list of nodes that form a path from node s to node t. Returns the empty list if no such path exists.

Examples

```
(define graph (graph-add-edges '() '(a a a b c c d e e f g g h) '(b c d c a e c d g h f e g)))
;; graph is ((h g) (g e) (g f) (f h) (e g) (e d) (d c) (c e) (c a) (b c) (a d) (a c) (a b))
(graph-remove-node graph 'c) ==> '((h g) (g e) (g f) (f h) (e g) (e d) (a d) (a b))
(graph-shortest-path graph 'c 'h) ==> '(c e g f h)
(graph-shortest-path graph 'a 'z) ==> '()
(graph-shortest-path graph 'c 'c) ==> '(c)
```

Augmented Binary Search Trees in Scheme

Description

You must write an augmented binary search tree data type in scheme. Since scheme is not object-oriented you will simply write a set of functions that define operations on binary search trees. For this assignment, we will represent an augmented binary search tree as a list. An augmented binary tree is defined below.

1. The empty list is the empty tree.
2. Otherwise a tree is a list of exactly four elements. The first element is the root value, the second element is the left sub-tree, the third element is the right sub-tree, and the fourth element is the size of the tree. Any node having the empty tree for both the left and right sub-trees is a leaf. The size of the empty tree is 0. The size of all other trees is one more than the sum of the sizes of the left and right subtrees.

Functions

You must write the following augmented binary search tree methods. Each method must have efficient runtime performance and function as a binary-search tree.

1. `(abst-create-empty)` Returns an empty augmented binary search tree.
2. `(abst-create root left right)` Returns an augmented binary search tree having the specified root value, with left-subtree left and right-subtree right.
3. `(abst-insert bst f x)` Returns the augmented binary search tree that results from inserting x into binary-searchtree bst. Function f is a predicate that accepts two elements of the type contained in the tree and returns true if the left operand is less than the right.
4. `(abst-contains bst f g x)`: Returns true if bst contains element x as defined by the predicate f and false otherwise. Function f is a predicate that accepts two elements of the type contained in the tree and returns true if the two elements are equal and false otherwise. Function g is a predicate that accepts two elements of the type contained in the tree and returns true if the left operand is less than the right.
5. `(abst-position bst f g x)`: Returns the position of x in an in-order listing of bst. Predicates f and g are identical to those described in `abst-contains`. This method must not hvae a linear runtime.
6. `(abst-nth bst n)`: Returns the nth element of bst in an in-order listing of bst. This method must not have a linear runtime.
7. `(abst-remove bst f g x)` Returns the augmented binary search tree representing bst after removing x where f and g are predicates as defined in bst-contains.
8. `(abst-pre-elements bst)` Returns the elements of bst in pre-order sequence.
9. `(abst-in-elements bst)` Returns the elements of bst in in-order sequence.
10. `(abst-post-elements bst)` Returns the elements of bst in post-order sequence.
11. `(list->abst xs f)` Returns the abst that results from adding each element in list xs to an empty bst in the order they occur in xs using f as the bst-insert predicate.

Hints

The **abst-position** algorithm is described in an ad-hoc language below:

```
function abst-position(tree, x)
  if tree is empty then return undefined
  else if x < tree.root-value then return abst-position(tree.left-subtree, x)
  else if x = tree.root-value AND tree.left-subtree is not null then return tree.left-subtree.size + 1
  else if x = tree.root-value AND tree.left-subtree is null then return 1
  else if x > tree.root-value AND tree.left-subtree is not null then return tree.left.size + 1 + abst-position( tree.right-subtree, x)
  else return 1 + abst-position( tree.right, x)
```

The **abst-select** algorithm is described in an ad-hoc language below:

```
function abst-select(tree, i)
  let VAL = tree.left.size + 1
  if i == VAL then return tree.root-value
  else if i < VAL then return abst-select( tree.left, i )
  else return abst-select( tree.right, i - VAL )
```

Simple Expressions

Description

A scheme expression is, in a sense, a pre-parsed abstract syntax tree. In this problem you will write a function that accepts a scheme expression and simplifies it into canonical form such that a more computationally efficient abstract syntax tree is produced. For this assignment, scheme expressions are defined in Figure 1.

1. A variable is a scheme expression.
2. A number is a scheme expression.
3. Assuming that E1 and E2 are scheme expressions, then so are.
   a. (+ E1 E2)
   b. (- E1 E2)
   c. (* E1 E2)
   d. (/ E1 E2)
4. The meaning of the four operators is identical with numeric addition, subtraction, multiplication and division as defined in the scheme language with the exception of the "ERROR" value (note below).

Figure 1. Scheme Expressions

Simplify

Write a function named `simplify` that takes a scheme expression and simplifies that expression by applying the simplification rules below.

1. X+0 → X for any X other than ERROR (also for 0+X)
2. X*0 → 0 for any X other than ERROR (also for 0*X)
3. X*1 → X for any X other than ERROR (also for 1*X)
4. X-0 → X for any X other than ERROR
5. X/1 → X for any X
6. X/0 → ERROR for any X
7. X/X → 1 for any X other than ERROR and 0 and 1
8. X○Y → Z if X and Y are numbers and Z is the result of applying ○
9. X-X → 0 for any X other than ERROR
10. X○Y → ERROR if either X or Y is an ERROR
11. 0/X → 0 for any X other than 0 and 1

Additionally, the following rules must be applied.

1. For any simplified scheme expression involving either multiplication or addition
   a. If the operands are both variables, they are ordered alphabetically.
   b. If the operands include a variable and a value, the variable precedes the value.
   c. If one operand is a sub-expression and the other is either a variable or value, the variable or value precedes the sub-expression.
   d. If the operands are both sub-expressions, the ordering of the sub-expressions follows the ordering of their operand as given in: *, +, -, /.
2. A simplified expression will have no sub-expression that can be simplified by application of one or more of the above rules.

Examples

```
(simplify '(+ (+ 3 5) (* (/ y 1) (+ x 0))))==>(+ 8 (* x y))
(simplify '(+ z (/ 53 0)))==>error
(simplify '(+ z a))==>(+ a z)
(simplify '12)==>12
(simplify '(+ (- 5 2) 9))==>12
(simplify '(* x a))==>(* a x)
(simplify '(/ (+ 3 a) (+ a 3)))==>1
(simplify '(+ a 3))==>(+ a 3)
(simplify '(+ 3 a))==>(+ a 3)
(simplify '(+ (- 1 a) 3))==>(+ 3 (- 1 a))
(simplify 'x)==>x
(simplify '(* (- (+ (+ (- y (* 2 c)) z) (/ (* (/ x 2) (+ 4 0)) (* (- c 4) (+ z a)))) (- (* (/ (+ a c) (+ b 0)) (* (+ y a) (* b 5))) (+ (/ (* 0 7) (/ 2 3)) (- z (+ 4 7))))) (+ (- (- (* (+ z 1) (- 5 2)) (+ (* b 0) (- a z))) (* (/ (* 0 3) (+ 0 y)) c)) (/ z (* (+ (+ c 5) c) (+ (/ 4 z) (/ c 2))))))==>(* (- 6 (- a z)) (/ z (* (+ c (+ c 5)) (+ (/ 4 z) (/ c 2))))) (- (+ z (- y (* c 2))) (- (* (+ (* b 5) (+ a y)) (/ (+ a c) b)) (- z 1))))
(simplify '(- (+ z (+ (+ a (+ z 2)) (- (- 7 x) (- 0 b))) (* (/ (/ a z) (- x 1)) y))) y))==>(- (+ z (+ (* y (/ (/ a z) (- x 1))) (+ (+ a (+ z 2)) (- (- 7 x) (- 0 b)))) y)
(simplify '(* (+ (/ (+ (* (+ (/ c (/ a 4)) (/ (- 4 b) a)) (* (- (+ 5 x) (/ y b)) (* (- z y) (/ 6 b)))) (* (+ (- (/ 2 b) (/ y z)) (+ (+ 4 x) (/ z z))) (/ (/ (+ x z) x) (+ (+ b z) (* 7 x))))) (/ (* (/ (/ (* 2 a) (/ b 6)) (/ (/ 6 6) x)) (+ b b)) c)))==>(* (/ (+ (* (/ c (/ a 4)) (/ (- 4 b) a)) (* (* (- z y) (/ 6 b)) (- (+ x 5) (/ y b)))) (* (+ (+ 1 (+ x 4)) (- (/ 2 b) (/ y z))) (/ (/ (+ x z) x) (+ (* x 7) (+ b z))))) (/ (+ (+ b b) (/ (/ (+ a 2) (/ b 6)) (/ 1 x))) c))
(simplify '(+ (/ (/ (+ (- (/ (+ x 7) (* 5 1)) (- (/ c a) (/ 7 y))) (+ (+ (* 2 c) (- 4 a)) (/ (/ 5 7) (- z c)))) (/ (/ (- (- 1 1) (+ z 6)) (- a (- 4 z))) (+ x c))) (+ x (+ (- (/ (* 7 3) (- 0 3)) (* a (/ 6 0))) (- (+ x a) (- y (- a 7))))))==>error
(simplify '(+ (/ (* (/ (+ (/ (- b (* x 6)) c) (- (- (* y 3) (+ z 0)) (- (- x 3) (- z 2)))) (+ (* (+ (* y 3) (* x 1)) (* (- 3 0) (/ y 3))) x)) (+ (+ a a) (- (* x b) (+ (* a a) (/ (- y b) (- 0 x)))))) (/ (- y b) (- 0 x)))))==>(+ (+ (+ a a) (- (* b x) (+ a (/ (- y b) (- 0 x))))) (/ (* (- (- (* y 3) z) (- (- x 3) (- z 2))) (/ (- b (* x 6)) c)) (+ x (+ (* 3 (/ y 3)) (+ x (* y 3)))))))
(simplify '(/ (/ (* (+ (* (/ a (* 2 7)) c) a) x) (+ (- (/ (/ (/ z 4) a) (- b (+ 5 3))) (/ (+ (/ x 5) (/ b y)) (- (+ c 4) (/ 1 6)))) (/ (+ (- (- a 2) a) (/ (- 2 a) b)) (/ (/ y a) b)))))==>(/ (/ (* a (+ c (/ a 14))) x) (+ (- (/ (/ (/ z 4) a) (- b 8)) (/ (+ (/ x 5) (/ b y)) (- (+ c 4) 1/6))) (/ (+ (- (- a 2) a) (/ (- 2 a) b)) (/ (/ y a) b))))
```

Grammars

Consider an expression language with two binary operators "$" and "#" and two unary prefix operators "*" and "@". It also includes a single alphabetic symbol "X" along with parenthesis "(" and ")". An EBNF expression grammar is given below where the starting non-terminal is <expr> and terminals are highlighted.

<expr> ::= <term> { `$` <term>}
<term> ::= <factor> { `#` <factor>}
<factor> ::= [`@` | `*`] (`X` | `(` <expr> `)`)

Problem

1. Give an equivalent unambiguous BNF grammar for this expression language.
2. Give the associativity of the two binary operators and a precedence table for all four operators in your BNF grammar.
3. Using your BNF specification, provide a parse tree for the expression `*X$X$@((*X)#X@*X)`.

Syntactic Ambiguity

Prove that the following BNF grammar is ambiguous.

```
N = {<S>, <A>, <I>}
T = {a, b, c, x}
P = {<S> ::= <A>
     <A> ::= <A><X><A>
     <A> ::= <I>
     <I> ::= a
     <I> ::= b
     <I> ::= c}
S = <S>
```

Submission

1. You must submit all your work using GitLab using a project named `CS421` with a folder named **hw2**. Here is a GitLab tutorial.
2. The scheme code must be placed in three files: **graph.rkt** for the disk scheduling functions, **abst.rkt** for the augmented binary search tree functions, and **simplify.rkt** for the simplication function. Each of these files must be contained in the **hw2** folder (not subfolders).