Trees and Grammars

Added a note to the simplification problem related to sorting symbols. 10/3
Updated the examples section of the simplify function. Several of the previous examples were incorrect with respect to ordering.10/10
I *really* fixed the examples this time...? 10/11

Binary Search Trees in Scheme

Description

You must write a 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 a binary search tree as a list. A binary tree is defined below.

1. The empty list is the empty tree.
2. Otherwise a tree is a list of exactly three elements. The first element is the root value, the second element is the left sub-tree and the third element is the right sub-tree.

Functions

You must write the following binary search tree methods. Each method must have a reasonably efficient runtime performance.

1. `(bst-create-empty)` Returns an empty binary search tree.
2. `(bst-create root left right)` Returns a binary search tree having the specified root value, with left-subtree left and right-subtree right.
3. `(bst-insert bst f x)` Returns the 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. `(bst-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. `(bst-remove bst f g x)` Returns the binary search tree representing bst after removing x where f and g are predicates as defined in bst-contains.
6. `(bst-pre-elements bst)` Returns the elements of bst in pre-order sequence.
7. `(bst-in-elements bst)` Returns the elements of bst in in-order sequence.
8. `(bst-post-elements bst)` Returns the elements of bst in post-order sequence.
9. `(list->bst xs f)` Returns the bst 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.

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))))) (+ (- (- (* (* 2 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 2) (/ 4 z)) (+ c (+ c 5))))) (- (+ z (- y (* c 2))) (- (* (* (* b 5) (+ a y)) (/ (+ a c) b))
(- z 11))))
(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)))==>(* (/ (* (+ b b) (/ (/ (* a 2) (/ b
6)) (/ 1 x))) c) (/ (+ (* (* (- z y) (/ 6 b)) (- (+ x 5) (/ y b))) (* (/ (- 4 b) a) (/ c (/ a 4)))) (* (+ (+ 1 (+ x 4)) (- (/ 2 b) (/ y z)))
(/ (/ (+ x z) x) (+ (* x 7) (+ b z))))))
(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 (/ y y)) (/ (- 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)) (/ (* (/ b y) (/ x 5)) (- (+ c 4)
1/6))) (/ (+ (- (- a 2) a) (/ (- 2 a) b)) (/ (/ y a) b))))
```

Footprint

Write a function named `footprint` that takes a scheme expression and returns an approximate memory footprint. The footprint is determined by 1) counting the unique integer literals in the expression and 2) counting the unique variables in the simplified expression. The footprint is then given as 4 times the sum of these counts.

Hint

Simplification requires ordering symbols. I recommend writing the following function; a function that accepts two symbols and returns **true** if the first symbol is less than the second.

```
(define (symbol<? s1 s2) (string<? (symbol->string s1) (symbol->string s2)))
```

Grammars

Consider an expression language with two binary operators "$" and "#" and two unary prefix operators "*" and "@" and a single alphabetic symbol "X". An EBNF expression grammar is given below where the starting non-terminal is <expr>.

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

Problem

1. Give an equivalent 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` .
4. Show that your BNF grammar is either unambiguous or ambiguous. If ambiguous, provide a formal proof. If unambiguous provide a rigorous justification.

Syntax Charts and BNF

For each of the two syntax charts in Figure 2, provide an Extended BNF definition that describes exactly the same language (where the alphabet is given as {0, 1}) and informally justify your answer.

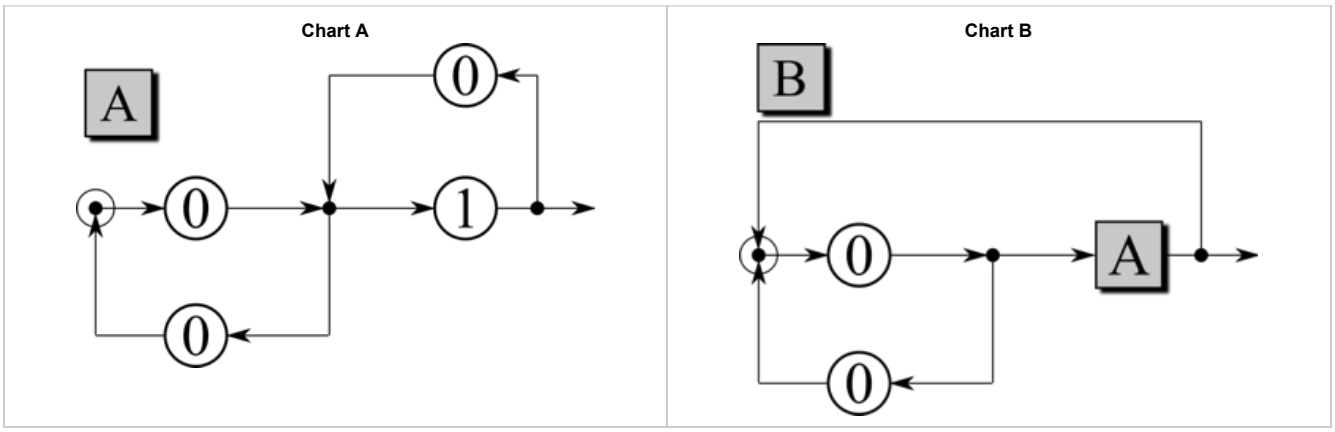

Figure 2. Syntax Charts

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(https://cs.uwlax.edu:17443) using a project named `cs421` with a folder named **hw2**. Here is a GitLab tutorial(http://charity.cs.uwlax.edu/shared/gitlab-tutorial/tutorial.html).
2. The scheme code must be placed in two files: one named **bst.scm** and one named **simplify.scm**. Of course, the bst.scm file contains all of the functions related to the binary-search-tree while the simplify.scm file contains the two functions `simplify` and `footprint`. Both of these files must be contained in the **hw2** folder.
3. The non-coding solutions must be placed in a PDF file named **hw2.pdf**. You can use a word-process or even take photos of hand-sketched solutions that are then packaged into a PDF file. Your solutions must be well organized and well formatted in addition to being correct. This file must be in the folder named **hw2**.