Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 4

Due: December 26, 2023, 11:55 PM

1 Introduction

In this assignment, you will implement a calculator using the Scheme programming language. This document provides detailed guidelines and requirements for the assignment.

2 The Task

Your task is to create various calculator procedures in Scheme. The assignment is structured into several parts, each building upon the previous. Here is a breakdown of the tasks:

2.1 Part 1: Two-Operator Calculator

Implement a procedure named `twoOperatorCalculator` which evaluates an infix expression containing only constant numbers, addition (`+`), and subtraction (`-`) operators. The input is a list, with operators and constants as separate elements.

- Example inputs and outputs:

```
1 ]=> (twoOperatorCalculator '(1))
;Value: 1

1 ]=> (twoOperatorCalculator '(1 + 2))
;Value: 3

1 ]=> (twoOperatorCalculator '(1 + -2))
;Value: -1

1 ]=> (twoOperatorCalculator '(2 - 1 + 3 + 1))
;Value: 5
```

```
1 ]=> (twoOperatorCalculator '(1 + 15 - 32/5 + -2))
; Value: 38/5

1 ]=> (twoOperatorCalculator '(32/5))
;Value: 32/5
```

Note: The expression `32/5` is considered a single number, not a division operation. However, if we slightly change the syntax and give the last example as

```
1 ]=> (twoOperatorCalculator '(32 / 5))
```

the behavior is not defined. You can consider that `twoOperatorCalculator` will never be applied to an expression in which there are division or multiplication operators and can assume that we will only test the procedure with valid inputs.

Similarly, in $(1 + -2)$, the symbol $-$ stands for the negative sign, not the subtraction operator. Whenever we use $-$ as a subtraction operator, it is given as a separate item in the list. Thus, when the above expression is given as input to the procedure the procedure must output -1. On the other hand, the behavior of the procedure for input like `(1 -2)` (i.e., there is no operator, and the operands are 1 and -2) is not defined, and as already said, you can assume that we will not test this procedure with such inputs (or invalid inputs in general).

2.2 Part 2: Four-Operator Calculator

Create a procedure `fourOperatorCalculator` to evaluate left-associative infix multiplication (`*`) and division (`/`) operations, and return a list of infix addition and subtraction.

- Example inputs and outputs:

```
1 ]=> (fourOperatorCalculator '(1))
;Value: (1)

1 ]=> (fourOperatorCalculator '(2 * 3))
;Value: (6)

1 ]=> (fourOperatorCalculator '(5 / 2))
;Value: (5/2)

1 ]=> (fourOperatorCalculator '(5 / 2 * 2))
;Value: (5)

1 ]=> (fourOperatorCalculator '(1 + 3 * 5 - 4 / 5 * 8 +
    2 * -1))
;Value: (1 + 15 - 32/5 + -2)
```

Note: You can assume that we will only test your procedure with valid inputs.

2.3 Part 3: Nested Calculations

Develop `calculatorNested` to handle operations within nested lists, with precedence given to operations in parentheses, and return a list that contains infix addition, subtraction, multiplication, and division operations.

- Example inputs and outputs:

```
1 ]=> (calculatorNested '(1 + (1 + 1 * 2) * 5 - 4 / 5 *
    8 + (5 - 3) * -1))
;Value: (1 + 3 * 5 - 4 / 5 * 8 + 2 * -1)

1 ]=> (calculatorNested '(1 + 2 * (2 - (3 + 5) / 2)))
;Value: (1 + 2 * -2)

1 ]=> (calculatorNested '((3 + 2) * (2 - 1) + (4 / (2 +
    2))))
;Value: (5 * 1 + 1)

1 ]=> (calculatorNested '((1 + 3) * (2 - 5 + (6 / 3)) -
    7))
;Value: (4 * -1 - 7)
```

Note: There can be nested lists inside other nested lists. In this case, calculatorNested should calculate all nested lists. Again, you can assume that we will only test your procedure with valid inputs.

2.4 Part 4: Operator Check

Implement `checkOperators` to verify if the input list contains only allowed operations (addition, subtraction, multiplication, division, and nested lists).

- Example inputs and outputs:

```
1 ]=> (checkOperators '(1))
;Value: #t

1 ]=> (checkOperators '((((1))))))
;Value: #t
```

```
1 ]=> (checkOperators '(1 + (1 + 1 * -2) * 5 - -4 / 5 *
    8 + (5 - 3) * -1))
;Value: #t

1 ]=> (checkOperators '(+ 1 3))
;Value: #f

1 ]=> (checkOperators '(1 < 3))
;Value: #f

1 ]=> (checkOperators '1)
;Value: #f
```

Note: For any invalid input your procedure should return `#f`.

2.5 Part 5: Full Calculator

Finally, create `calculator` to compute infix operations with precedence rules. The procedure should first check whether the given list is syntactically correct. If the list is not syntactically correct, then the procedure should return `#f`, otherwise, it should calculate the operations and give the output as a result. Below is a sample output for this procedure:

- Example inputs and outputs:

```
1 ]=> (calculator '(1))
;Value: 1

1 ]=> (calculator '(1 + (1 + 1 * 2) * 5 - 4 / 5 * 8 +
   (5 - 3) * -1))
;Value: 38/5

1 ]=> (calculator '(2 * (1 + (1 + (1 + (1 + (1 + ((1)))
   ))))))
;Value: 12

1 ]=> (calculator '(1 -2))
;Value: #f

1 ]=> (calculator '(1 < 3))
;Value: #f

1 ]=> (calculator '1)
;Value: #f
```

Note: Again, just like the `checkOperators`, for any invalid input your procedure should return `#f`.

3 How to Submit

Submit your Scheme file named `username-hw4.scm` where username is your SUCourse username. We will test your submissions in the following manner. A set of test cases will be created to assess the correctness of your scheme calculator. Each test case will be automatically appended to your file. Then the following command will be executed to generate an output. Then your output will be compared against the desired output.

```
scheme < username-hw4.scm
```

So, make sure that the above command is enough to produce the desired output.

4 Notes

- **Important**: Name your files as you are told (pay extra attention to the extension of the file you submit) and **don't zip them.** [-10 points otherwise]
- **Important: Make sure your procedure names are exactly the same as it is supposed to be (i.e., case sensitive)!**
- **Important: Important: Since this homework is evaluated automatically make sure your output is exactly as it is supposed to be.**
- No homework will be accepted if it is not submitted using SUCourse+.
- You may get help from our TA or from your friends. However, **you must implement the homework by yourself**.
- Start working on the homework immediately.
- If you develop your code or create your test files on your own computer (not on flow.sabanciuniv.edu), there can be incompatibilities once you transfer them to flow.sabanciuniv.edu. Since the grading will be done automatically on flow.sabanciuniv.edu, we strongly encourage you to do your development on flow.sabanciuniv.edu, or at least test your code on flow.sabanciuniv.edu before submitting it. If you prefer not to test your implementation on flow.sabanciuniv.edu, this means you accept to take the risks of incompatibility. Even if you may have spent hours on the homework, you can easily get 0 due to such incompatibilities.
- LATE SUBMISSION POLICY:
  Late submission is allowed subject to the following conditions:

- Your homework grade will be decided by multiplying what you get from the test cases by a "submission time factor (STF)".
- If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
- If you submit late, you will lose 0.01 of your STF for every 5 mins of delay.
- We will not accept any homework later than 500 minutes after the deadline.
- SUCourse+'s timestamp will be used for STF computation.
- If you submit multiple times, the last submission time will be used.