Homework 3: Continuations

Due date: October 20 (Wednesday) at 11:59pm

Addendum: Partners allowed

Going forward, we will optionally allow you to work in pairs for the homework assignments.

Every student should navigate navigate to Canvas → People, then click the "hw3-groups" tab and sign up for a group with your partner (or alone if you choose to work alone). One member of each group should submit the assignment on Canvas.

Overview

This homework consists of two parts:

1. Using continuations, implement `throw` and `try_except` functions that allow programs to throw and handle exceptions. Then, using the two functions, implement a simple error-detecting calculator.
2. Using continuations, implement `attempt` and `assert` functions that allow programs to write backtracking algorithms. Then, using the two functions, implement a simple solver for a Sudoku puzzle.

Some important notes:

- You will use the Racket language, which natively supports continuations. The particular version of Racket you should use is **Racket Minimal 8.2**. You may either install it on your local machine or work on `myth.stanford.edu`. For the first option, download the "Minimal Racket" installation file corresponding to your system from the official webpage[1]. For the second option, you can access the Racket executable by adding the CS 242 directory to your search path. To do this, whenever you SSH into `myth` to work on the assignment, first run this command:

  ```
  export PATH=/afs/ir/class/cs242/tools/racket/bin:$PATH
  ```

  After executing this line, you should be able to run the command `racket`. You can also add this line to the bottom of the `.bashrc` file in your home directory, which is executed automiatcally every time you log in to `myth`.

  If you are using a non-default shell on `myth`, the command to add the CS 242 directory to your path may be different.

- You should read the following guide to Racket before starting the homework.

  `https://cs242.stanford.edu/materials/assignments/racket-guide.html`

  For more details on Racket, see `https://docs.racket-lang.org/guide/index.html`. We highlight a particularly important aspect of Racket that may be confusing at first: **be careful about parentheses**, they play a significant syntactic role in Racket and you should use them only when they are necessary. The CA's solution is written exclusively with the following built-in syntax or functions of Racket: `call/cc`, `define`, `lambda`, `let*`, `if`, `cond`, `else`, `printf`, `equal?`, `not`, `and`, `number?`, `>`, `+`, `-`, `*`, `/`, `list`, `first`, `second`, `third`, `rest`, `length`, and `range`.

[1] `https://download.racket-lang.org/releases/8.2/`

- You should not use any built-in syntax or functions of Racket to throw or handle exceptions. **To allow us to automatically enforce this restriction, the words "raise" and "handler" should not appear in any parts of your solution files (`prob*.rkt`), including comments.**

How to Use Continuations in Racket

In Racket, we can obtain and pass the current continuation to a function using the `call/cc` syntax. For example, the expression $(2 + (\text{call/cc } \lambda k.\, 3 + (\text{resume } k\ 4)))$ written in the language discussed in the lecture can be translated into Racket as follows.

$$\Bigg(\texttt{+ 2 } \Bigg(\texttt{call/cc } \Big(\texttt{lambda (k) (+ 3 (k 4))}\Big)\Bigg)\Bigg)$$

Here we use different sizes of brackets just to make the expression more readable. Note that $(\text{call/cc } \lambda k.\, e)$ is translated into Racket as `(call/cc (lambda (k) e))`, and $(\text{resume } k\ e)$ as `(k e)`. In this expression, the continuation of the subexpression `(call/cc ...)` is $(\lambda\square.(\texttt{+ 2 }\square))$, when represented as a lambda abstraction, since the subexpression is in the place of $\square$ in the entire expression $\texttt{(+ 2 }\square\texttt{)}$. What the `call/cc` syntax inside the expression does is as follows: the `call/cc` applies its argument `(lambda (k) (...))` to the continuation $(\lambda\square.(\texttt{+ 2 }\square))$ of the subexpression `(call/cc ...)` and then continues evaluation. This leads to the following evaluation steps, where the subexpressions being evaluated at each step are underlined.

$$\begin{aligned}
&\Bigg(\texttt{+ 2 } \underline{\Bigg(\texttt{call/cc } \Big(\texttt{lambda (k) (+ 3 (k 4))}\Big)\Bigg)}\Bigg)\\
\to\ &\Bigg(\texttt{+ 2 } \underline{\Bigg(\Big(\texttt{lambda (k) (+ 3 (k 4))}\Big)\ (\lambda\square.(\texttt{+ 2 }\square))\Bigg)}\Bigg)\\
\to\ &\Bigg(\texttt{+ 2 (+ 3 } \underline{((\lambda\square.(\texttt{+ 2 }\square))\ \texttt{4})}\texttt{)}\Bigg)\\
\to\ &\underline{\texttt{(+ 2 4)}}\\
\to\ &6
\end{aligned}$$

Note that the second last expression is obtained from the third last one, since `4` is applied to the continuation $(\lambda\square.(\texttt{+ 2 }\square))$ (i.e., the continuation $(\lambda\square.(\texttt{+ 2 }\square))$ is resumed with the value `4`). This way we can obtain and pass the current continuation using the `call/cc` syntax, and resume a continuation `k` with the evaluation result of `e` using `(k e)`. In this homework, you will figure out how to use `call/cc` appropriately to implement exception handling and backtracking algorithms.

Part 1: Exception handling

In the first part of this assignment, you will implement two functions `throw` and `try_except` using `call/cc` (Problem 1), which can serve as a building block for raising and handling exceptions, and then you will implement a simple error-detecting calculator using the two functions (Problem 2).

Problem 1: `throw` and `try_except`

The goal of this problem is to implement two functions `throw` and `try_except`, which correspond roughly to the `raise` keyword and the `try: ... except: ...` statement in Python. For example, `throw` and `try_except` should behave as follows.

Example program:

```
(try_except
  (lambda ()
    (printf "try: start\n")
    (throw "NameError")
    (printf "try: end\n"))
  (lambda (msg)
    (printf "except: start\n")
    (printf "except: ~a\n" msg)
    (printf "except: end\n")))
```

Expected print-out:

```
try: start
except: start
except: NameError
except: end
```

That is, `(throw msg)` should throw (or raise) an exception that carries a message `msg`. On the other hand, `(try_except try_f except_f)` should run `(try_f)` and, if an exception is thrown by `(throw msg)` during the evaluation of `(try_f)`, it should catch (or handle) the exception and run `(except_f msg)` immediately after the exception is thrown.

You can assume that `throw` takes a single string as an argument and `try_except` takes two functions `try_f` and `except_f` as arguments. To simplify the problem, assume that multiple uses of `try_except`'s cannot be nested and continuations cannot be created, stored, or called anywhere except in the definition of `throw` and `try_except`. If an exception is thrown by `throw` outside `try_f` (i.e., either inside `except_f` or outside `(try_except ...)`), you should print out an error message `ThrowError` and exit to the top-level.

Implement `throw` and `try_except` in `prob1.rkt` using `call/cc`. You can test your implementation by running `racket test/prob1_test.rkt`. The expected outputs are provided in the test file. It will be useful to utilize the stack defined in `prob1.rkt` to store continuations and some arguments of `try_except`, and the function `exit` to exit to the top-level. The CA's solution without comments is 20 lines of code.

Hint: One way to implement `throw` and `try_except` is as follows. Note that the following description is not complete (just gives you an idea) and there can be other ways to implement them. In `try_except` which takes arguments `try_f` and `except_f`, push the current continuation `k` and `except_f` to the stack, call `(try_f)`, and then pop `k` and `except_f` from the stack. In `throw` which takes an argument `msg`, pop the topmost `k` and `except_f` from the stack if the stack is not empty, call `(except_f msg)`, and then resume `k` with the return value of the call.

Problem 2: Error-detecting calculator

The goal of this problem is to implement a function `eval`, which takes an arithmetic expression as an argument, and prints out an error message if the expression contains an error of certain types, and prints out the evaluation result otherwise. For example, `eval` should behave as follows.

Example program:

```
(eval (list "+" 1 (list "/" 2 (list "-" 3 3))))
(eval (list "+" 1 (list "#" 2 3)))
(eval (list "+" 1 (list "*" 2 3)))
```

Expected print-out:

```
DivError
OpError
7
```

The first line prints out a division-by-zero error, as we have to compute `(/ 2 0)` while evaluating the given expression. The second line prints out an invalid operator error, as the given expression includes an invalid operator `#`. The third line prints out the evaluated result, as the given expression makes no errors.

You can assume `eval` takes a single argument of the following form, where n and s denote a number and a string in Racket, respectively.

$$e ::= n \mid (\texttt{list}\ s\ e\ e)$$

The function `eval` should detect and print out the two types of errors (and no more): a division-by-zero error, which occurs when an argument to `eval` contains a division-by-zero computation, and an invalid operator error, which occurs when an argument uses an operator other than `"+"`, `"-"`, `"*"`, and `"/"`. If no errors of the two types are found, `eval` should print out the evaluated result. You can assume any argument to `eval` has at most one error.

Implement `eval` in `prob2.rkt` using `throw` and `try_except`. You can test your implementation by running `racket test/prob2_test.rkt`. The expected outputs are provided in the test file. The CA's solution without comments is 22 lines of code.

Part 2: Backtracking algorithm

In the second part of this assignment, you will implement two functions `attempt` and `assert` using `call/cc` (Problem 3), which can serve as a building block for writing backtracking algorithms, and then you will implement a simple Sudoku solver using the two functions (Problem 4).

Problem 3: `attempt` and `assert`

The goal of this problem is to implement two functions `attempt` and `assert`. The function `attempt` takes a list `l` of choices as an argument and tries out each choice in `l` in order, until every argument of all subsequent `assert`s becomes true, or until the `attempt` tries out all the choices listed in `l`, whichever happens first. For example, `attempt` and `assert` should behave as follows.

Example program:

```
(let* ([x (attempt (list 1 2 3))]
       [y (attempt (list 5 6 7))]
       [_ (printf "x=~a, y=~a\n" x y)]
       [b (assert (equal? 15 (* x y)))])
  (printf "res: x=~a, y=~a, b=~a\n" x y b))
```

Expected print-out:

```
x=1, y=5
x=1, y=6
x=1, y=7
x=2, y=5
x=2, y=6
x=2, y=7
x=3, y=5
res: x=3, y=5, b=#t
```

In the first and second lines, the program uses `attempt` to try out three choices (`1`, `2`, and `3`) for the variable `x` and another three choices (`5`, `6`, and `7`) for the variable `y`. In the fourth line, the program uses `assert` to describe a condition that needs to be satisfied, namely `(* x y)` should be equal to `15`. From the expected print-out, you can check that the two calls to `attempt` indeed try out each of their choices one-by-one sequentially, until the condition of the `assert` becomes true. Note that the two calls to `attempt` should try their choices exactly in the order shown in the above expected print-out.

You can assume that `attempt` takes a single non-empty list as an argument and `assert` takes a single expression that evaluates to a boolean. The expression `(attempt l)` should return one of the elements of `l` in the specific order described above. The expression `(assert e)` should return `#t` if `e` evaluates to `#t` at some point (possibly after trying out some choices), and should return `#f` if `e` evaluates to `#f` for all choices of the prior `attempt`s.

Implement `attempt` and `assert` in `prob3.rkt` using `call/cc`. You can test your implementation by running `racket test/prob3_testN.rkt`, where $1 \leq$ `N` ≤ 3. The expected outputs are provided in the test file. It will be useful to utilize the stack defined in `prob3.rkt` to store continuations and some parts of the argument of `attempt`. The CA's solution without comments is 18 lines of code.

Problem 4: Sudoku solver

The goal of this problem is to implement a function `solve`, which solves a given Sudoku puzzle by a simple backtracking algorithm. To explain a Sudoku puzzle, define a *state* as a 9×9 grid partially (or fully) filled with the numbers $1, \cdots, 9$. We say a state is *valid* if each number in the state does not appear twice in each row, each column, and each of the nine 3×3 subgrids. A Sudoku puzzle starts with an initial state, and the objective of the puzzle is to find a valid state that extends the initial state and is fully filled. Such a valid state is a solution of the puzzle.

You can assume that `solve` takes a single argument `state`, which represents an initial state of a Sudoku puzzle, and that `state` has at most one solution. The expression `(solve state)` should return a solution to `state` if it exists, and should return `#f` otherwise. Any state (e.g., an argument or a return value of `solve`) should be represented as a list of lists in the following way: for example, `(list (list 0 3 9) (list 8 0 7))` represents the state where 9 is written at the intersection of row 0 and column 3, and 7 is written at the intersection of row 8 and column 0.

Implement `solve` in `prob4.rkt` using `attempt` and `assert`. You can test your implementation by running `racket test/prob4_testN.rkt`, where $1 \leq$ `N` ≤ 3. The expected outputs are provided in the test file. It will be useful to utilize some helper functions defined in `prob4.rkt` to perform several primitive operations on states. The CA's solution without comments is 16 lines of code. Your solution should run in 15 seconds on the myth cluster for each `test/prob4_testN.rkt`; the CA's solution runs in < 2 seconds on the myth cluster for each testfile.

Submission

- Regardless of whether you are working in a pair, navigate to Canvas → People, then click on the "hw3-groups" tab and sign up for a hw3 group. If you are working with a partner, make sure you both sign up for the same group.
- Edit `README.txt` to include both your student ID number (the last 8-digit number) and your partner's student ID number (if you are working in pair).
- **Generate `solution.tar.gz` by running `python3.6 src/submit.py`, and one member of each group should upload the tarball file to Canvas.** Make sure the script gives no errors or warnings.
- Make sure the tarball file consists precisely of: `prob{1,2,3,4}.rkt` and `README.txt`.