Project Higher-Order Programming 2019–2020: Breaking Loops

Lecturer: Prof. Dr. Jens Nicolay
Assistant: Dr. Steven Keuchel

The deadline for this project is **Sunday 12th of January 2020** at 23:59.
All deliverables should be uploaded via Canvas.
The project should be made individually. Plagiarism is not tolerated.
Questions about this assignment can be sent to steven.keuchel@vub.be.

1 Assignment

During the course you have seen an implementation of the non-analyzing meta-circular Scheme evaluator in continuation-passing style (CPS). You can download this evaluator from the exercise website[1]

The goal of this assignment is to extend this implementation with a new feature that is not available in regular Scheme: `while` loops that support `break` and `continue`.

You are certainly familiar with loops in imperative languages, where they form the prevalent method to iterate over collections and to implement general recursion. The project proceeds in two stages: you implement a basic `while` special form first and then extend it to support `break` and `continue`.

1.1 Basic `while` loops

While loops use a predicate that stipulates if the loop keeps iterating or if the loops should be exited. Extend the meta-circular evaluator with a new special form:

```
(while <pred> <exp_1> .. <exp_n>)
```

The `while` form takes at least one argument `<pred>` that represents the iteration predicate. The remaining – possibly zero – arguments `<exp_1> .. <exp_n>` form the body of the loop and are evaluated in sequence.

[1] `https://soft.vub.ac.be/~skeuchel/hop/scheme-interpreters/04-cps/04-cps.scm`

Example The following code snippet exemplifies the basic use of `while`. The `sum` procedure iterates over the list of numbers `l` and accumulates the sum in the local binding `s`. The loop terminates when the end of the list is reached.

```
(define (sum l)
  (define s 0)
  (while
    (not (null? l))
    (set! s (+ s (car l)))
    (set! l (cdr l)))
  s)
(sum '(1 2 3)) ;; --> 6
```

Example One of the questions is, what the return value of the `while` form should be, in particular when the loop body was never executed. For consistency, you should simply always return a special symbol (or the void value):

```
(while #f) ;; --> 'ok
```

You can either implement the basic while loop *directly* or by *desugaring*. But the extension to `break` and `continue` you will have to implement directly without desugaring.

1.2 Break and continue

Break allows the programmer to exit the loop from a nested position and continue skips the rest of the current iteration and starts with the next iteration.

Extend your language with two new parameterless expression forms:

`(break)` and `(continue)`.

You can either implement one of these first and then the other, or implement both at the same time. **Implement these directly without desugaring.**

1.2.1 Break

The intended behaviour of `break` is that the enclosing loop is exited immediately without evaluating the predicate for the next iteration. Furthermore, `break` can be called from a nested position and does not have to be a direct sub-expression of a `while` loop. If a `break` statement is enclosed by multiple `while` loops, the innermost loop is exited.

Example The following code implements the function `drop-while` that drops the prefix of a list that fulfills the given predicate and returns the rest. Note that this is not a filter function: the element 15 is not dropped for the example call to `drop-while`.

```
(define (drop-while p l)
  (while
    (not (null? l))
    (if (p (car l))
      (set! l (cdr l))
      (break))
  l)
(drop-while odd? '(1 3 6 10 15)) ;; --> (6 10 15)
```

Example The `break` and `continue` forms should only be used in conjunction with `while` loops and should not for instance return from a function or similar. If no valid `while` loop is enclosing an invocation of `break` or `continue`, a runtime error messages should be signaled.

```
  (break) ;; --> ERR: Not inside a while loop.
```

1.2.2 Continue

The intended behaviour of `continue` is that the next iteration of the innermost enclosing loop is started. The actual execution of the loop body for the next iteration is still contingent on the result of the predicate, i.e. if after executing `continue` the predicate evaluates to false, the loop body is not entered and the loop is exited.

In the example below, the `continue` statement is always called and the dangerous remainder of the loop is never executed. Furthermore, the predicate is re-evaluated for each iteration even after `continue` as can be seen by the displayed output `123456`.

```
(define (display-seq i n)
  (while
    (begin (display i) (< i n))
    (set! i (+ i 1))
    (continue)
    (display "launch missiles")))
(display-seq 1 6) ;; --> 123456
```

1.2.3 Design choices

When implementing `break` and `continue` you will have to make certain design choices:

- What should the behaviour be when calling `break` or `continue` inside a `while` predicate,

  ```
  (while <pred> (while (break)))
  ```

  This does not occur in most C-like languages, because of a stratification of terms into expressions and statements.

- What should the behaviour be when a lambda is separating a `while` loop from a `break`?

  ```
  (while <pred> ((lambda () (break))))
  ```

- What is the return value of a `while`-loop that was exited due to `break` or after a `continue` with a false predicate?

Document and motivate your choices and present meaningful examples that highlight them.

2 Evaluation and Delivery

This assignment accounts for 50% of your final mark and is required in order to pass the course as a whole.

You need to hand in the following.

1. The source code of your implementation. Be sure to include all necessary files if you have split your solution into multiple files!
2. A report (±6 pages, in pdf) that provides an overview of the design decisions you made during the project, and difficulties that you encountered (and how you overcame them). If you typeset your report in LaTeX, you you can use the minted package[2] for code highlighting. Typesetting in LaTeX is not a requirement.
3. An example program with multiple functions that highlight the behaviour of your application. You can refer to these examples in your report.

Include all requested files into a single zip file with the name `hop-project-<netid>.zip`, and submit this file on Canvas before the deadline.

Your project will be graded by looking at both the required functionality, as well as the code quality of your implementation.

[2] `https://github.com/gpoore/minted`