Scheme Continuations and Macros

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

September 30, 2020

Table of Contents

Table of Contents

Continuations and macros form some of Scheme's more advanced features. While they are not strictly necessary to be productive in Scheme, they are very powerful features.

The syntax of neither continuations nor macros will be on the midterm. However, I may ask about the motivation behind *why* macros are needed.

Table of Contents

Continuations

The R6RS standard says, “Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation.”

Continuation Example

Given the expression

```
(+ 1 3)
```

The continuation of 3 in the above expression adds 1 to it.

Continuation Definition

Definition (Continuation)

The continuation of an expression E is the expression that wants the evaluated result of E.

Continuations

- Usually we don't need to think about continuations when we code in Scheme.

Continuations

- Usually we don't need to think about continuations when we code in Scheme.
- However, there are rare occasions where we want to be able to deal with continuations manually.

call-with-current-continuation

- Often abbreviated `call/cc` in many Scheme implementations, including Racket.
- Syntax: `(call/cc fn)`
- `call/cc` calls `fn`, a function that has one parameter, with the argument being an *escape procedure*.
- According to the R6RS standard, "The escape procedure can then be called with an argument that becomes the result of the call to `call/cc`. That is, the escape procedure abandons its own continuation and reinstates the continuation of the call to `call/cc`."

Example of `call-with-current-configuration`

```
(let ((fn (lambda (escape)
            (+ 2 (escape 3)))))
  (+ 1 (call/cc fn)))
```

Example of `call-with-current-configuration`

```
(let ((fn (lambda (escape)
            (+ 2 (escape 3)))))
  (+ 1 (call/cc fn)))
```

The above code evaluates to 4, because `(escape 3)` is performing $1 + 3$, and because the continuation of `call/cc fn)` is $+1$.

Demo in DrRacket

Notes about `call-with-current-configuration`

- `fn` in `(call/cc fn)` always takes one argument: the escape procedure.
- The escape procedure's parameters are the same number as the continuation of the call to `call/cc`.
- The escape procedure is a closure that can be called at any time. It can be passed along like any other value, and it can be stored like any other value.

More about Continuations

Because the escape procedure can be called at any time by any expression that has access to it, continuations can be used to implement a wide variety of custom control-flow operations. Use continuations with care.

Table of Contents

Motivation for Macros

We can extend Lisp by defining functions that are based on existing functions:

```
(define (sqrt n)
  (expt n 0.5))

(sqrt 2)
```

Limitations of Function Definitions

However, there are some constructs we would like to provide that would be difficult to express as functions.

Implementing `let`

Suppose we were implementing our own `let` function. Below is a reasonable attempt:

```
(define (my-let bindings body)
  (if (empty? bindings)
      body
      ((lambda (first (first bindings))
         (my-let (rest bindings) body))
        (second (first bindings)))))
```

Implementing `let`

Suppose we were implementing our own `let` function. Below is a reasonable attempt:

```
(define (my-let bindings body)
  (if (empty? bindings)
      body
      ((lambda (first (first bindings))
         (my-let (rest bindings) body))
        (second (first bindings)))))
```

What is wrong with this approach?

The problem is that all function arguments must be evaluated unless they are quoted.

Implementing `let`

We could sidestep the problem by requiring the quoting of arguments we don't want evaluated and then using the `eval` function inside of `my-let` in order to provide more fine-grained control over evaluation:

```
(my-let (('x 5)
         ('y 10))
  '(+ x y))
```

Implementing `let`

We could sidestep the problem by requiring the quoting of arguments we don't want evaluated and then using the `eval` function inside of `my-let` in order to provide more fine-grained control over evaluation:

```
(my-let (('x 5)
         ('y 10))
  '(+ x y))
```

However, it would be inconvenient for programmers if they were required to quote so many arguments like this.

Solution to Implementing `let`: Macros

The solution is to use macros, which will give us finer control over how a Scheme expression is evaluated without resorting to quoting.

Solution to Implementing `let`

```
; Solution is from Veit Heller's blog at
; blog.veitheller.de/Scheme_Macros_III:_Defining_let.html
(define-syntax my-let
  (syntax-rules ()
    ((my-let ((var val) ...) body ...)
     ((lambda (var ...) body ...) val ...))))
```

Demo in DrRacket

Scheme and Racket have a variety of mechanisms for defining macros, some of them implementation-dependent. We will show examples of `define-syntax` and `define-syntax-rule` in Racket.

define-syntax-rule Example in Racket

```
; From Professor Thomas Austin's CS 152 Slides
(define-syntax-rule (swap x y)
  (let ((tmp x))
    (set! x y)
    (set! y tmp)))
```

define-syntax-rule Example in Racket

```
; From Professor Thomas Austin's CS 152 Slides
(define-syntax-rule (my-if c thn els)
  (cond ((and (list? c) (empty? c)) els)
        ((and (number? c) (= 0 c)) els)
        ((and (boolean? c) (not c)) els)
        (else thn)))
```

For more information about macros, please see the documentation for `define-syntax` and `define-syntax-rule` in Racket.

Summary of Macros

- Macros are necessary for implementing certain language features in Scheme without excessive quoting.
- There are many different mechanisms for implementing macros in Scheme and Racket, but some basic functions for implementing them in Racket are `define-syntax` and `define-syntax-rule`.

Table of Contents

Summary of Topics to Study

- Differences between procedural, structured, and functional programming
- Regular and context-free grammars
- Abstract syntax trees
- Scheme features covered in class (except for `call/cc` and those related to macro creation)
- Using tail recursion
- Environments
- Evaluating Expressions with Environments (note that you won't need your Project 1 code)

Regular and Context-Free Grammars

Please be prepared to:

1. Know the differences between these two types of grammars.
2. Be able to recognize code that conforms to these grammars and code that does not conform.
3. Be familiar with Backus-Naur Form (BNF) and its extended variant EBNF.
4. Be able to describe how operator precedence works in a EBNF grammar.
5. Be able to draw an abstract syntax tree given a language form and a grammar.

Environments and Evaluation

Please be prepared to describe in full, complete detail with diagrams how expressions like this are evaluated:

```
(let* ((x 3)
       (y (* x x)))
  (+ x y))
```

Words of Advice

Because this is a take-home exam, the exam won't be about definitions and other facts that can be simply looked up; it will be about how well you have mastered the concepts taught so far in CS 152 and how you can apply what you have learned.

For example, don't just memorize the differences between a regular grammar and a context-free grammar. Study *why* this matters.