State, Mutation, and Environments

Michael McThrow

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

September 16, 2020

Table of Contents

Table of Contents

State, Mutation, and Environments

What is State?

- A program's *state* is essentially what a program "memorizes" throughout its lifetime.

What is State?

- A program's *state* is essentially what a program "memorizes" throughout its lifetime.
- A program's state may be located in local and global variables, function parameters, data referenced by pointers, and even external sources (such as file systems, databases, Web servers, etc.).

Functional Programming and State

- One goal of functional programming style is to make programs more mathematically tractable by emphasizing *immutability* and *reducing side effects*.

Functional Programming and State

- One goal of functional programming style is to make programs more mathematically tractable by emphasizing *immutability* and *reducing side effects*.
- A *side effect* means modifying the state outside of the current scope.

Functional Programming and State

- One goal of functional programming style is to make programs more mathematically tractable by emphasizing *immutability* and *reducing side effects*.
- A *side effect* means modifying the state outside of the current scope.
- Examples of side effects:

Functional Programming and State

- One goal of functional programming style is to make programs more mathematically tractable by emphasizing *immutability* and *reducing side effects*.
- A *side effect* means modifying the state outside of the current scope.
- Examples of side effects:
  - Modifying a global variable outside of global scope.

Functional Programming and State

- One goal of functional programming style is to make programs more mathematically tractable by emphasizing *immutability* and *reducing side effects*.
- A *side effect* means modifying the state outside of the current scope.
- Examples of side effects:
  - Modifying a global variable outside of global scope.
  - Updating a database that the program is connected to.

Functional Programming and State

- One goal of functional programming style is to make programs more mathematically tractable by emphasizing *immutability* and *reducing side effects*.
- A *side effect* means modifying the state outside of the current scope.
- Examples of side effects:
  - Modifying a global variable outside of global scope.
  - Updating a database that the program is connected to.
  - Accessing I/O devices (this includes reading)

However, certain tasks would be difficult to perform without mutation and side effects.

Variable Mutation in Scheme

- Scheme provides the `set!` function for mutating a variable.
- It does not matter if the variable is local or global; `set!` will modify it.
  - Note that it is conventional in Scheme for functions that perform mutation to end in `!`.
- Example: `(set! x 10)`
  - Equivalent to `x = 10` in procedural programming languages.

Mutating cons Cells in Scheme

Scheme (but not Racket) provides the functions `set-car!` and `set-cdr!` for modifying the first and second elements of a cons cell, respectively.

Mutating cons Cells in Scheme

Scheme (but not Racket) provides the functions `set-car!` and `set-cdr!` for modifying the first and second elements of a cons cell, respectively.

Example:

```
#!r6rs
(import (rnrs) (rnrs mutable-pairs))
(define my-list (cons 1 '()))
(set-car! my-list 5)
(set-cdr! my-list (cons 2 (cons 3 '())))
; my-list now evaluates to '(5 2 3)
```

A Reminder for CS 152's Scheme Assignments

We will be programming in functional style; therefore, `set!`, `set-car!`, `set-cdr!`, and other mutation functions are banned in this course.

Table of Contents

For the next few lectures, we will cover how to write interpreters that evaluate Scheme expressions.

In Lab 1, you wrote evalators for arithmetic expressions written in postfix (e.g., `3 5 * 10 -`) and infix notation (e.g., `3 * 5 - 10`).

Suppose we extended the infix calculator to support variables:

```
Calculator> x = 5
x = 5
Calculator> 2 * x
10
```

Suppose we extended the infix calculator to support variables:

```
Calculator> x = 5
x = 5
Calculator> 2 * x
10
```

How would we implement this?

- The calculator could maintain a data structure that maps variable names to values.

- The calculator could maintain a data structure that maps variable names to values. (`x = 5` results in storing key `x` with value 5 in the map.)

- The calculator could maintain a data structure that maps variable names to values. (`x = 5` results in storing key `x` with value 5 in the map.)
- When the interpreter visits a variable name (e.g., when evaluating `x`), it then searches the map for a key with that name.
  - If there is a key, the visitor returns the value associated with that key (e.g., `2 * x` becomes `2 * 5`, which evaluates to 10).
  - If there is no key, the calculator returns an error.

Now, how do we deal with the introduction of user-defined functions to our calculator?

```
# sqrt.calc -- Script for executing square root function
function sqrt(x)
  return x ^ 0.5
end
x = sqrt(5)
```

```
# sqrt.calc -- Script for executing square root function
function sqrt(x)
  return x ^ 0.5
end
x = sqrt(5)
```

Some implementation challenges arise, namely:

```
# sqrt.calc -- Script for executing square root function
function sqrt(x)
  return x ^ 0.5
end
x = sqrt(5)
```

Some implementation challenges arise, namely:

- How do we define the function?

```
# sqrt.calc -- Script for executing square root function
function sqrt(x)
  return x ^ 0.5
end
x = sqrt(5)
```

Some implementation challenges arise, namely:

- How do we define the function?
- How do we make sure that different scopes do not conflict with each other?

```
# sqrt.calc -- Script for executing square root function
function sqrt(x)
  return x ^ 0.5
end
x = sqrt(5)
```

Some implementation challenges arise, namely:

- How do we define the function?
- How do we make sure that different scopes do not conflict with each other?

A simple map is not going to do anymore. Instead, we maintain *environments* in order to deal with these scoping issues.

Environments

- An environment is associated with a *frame*.
- The interpreter maintains a mapping of environment names to frames.
- A frame is a data structure that consists of two elements:
  1. A mapping of names to values.
  2. The name of the *enclosing environment*, i.e., the environment which scope encompasses this environment.
- The *global environment* is the top-most environment in the program; nothing encompasses it. The global environment also contains definitions of system-defined names.
- An environment corresponds to a level of scope in the program.

Examples of Environments (in supplemental slide deck)

Key Takeaways about Environments

- Evaluating symbols such as `x` and `PI` require looking up environments, starting from the current environment and potentially searching each encompassing environment.
- The `define` primitive function assigns names to values and places them into the current environment.
- Evaluating a function call creates a new environment.
- Calling a function binds function parameters to the arguments of the function call in the calling environment.
- Environments only last for as long as its corresponding function call is still running. The only exception is the global environment, which lasts for as long as the interpreter runs.
- The global environment contains names of primitive functions that are built into the language that cannot be expressed as libraries.

What Are Closures?

A `closure` is a higher-order function that has access to the environment in which it was created.

Example of a Closure

```
; Borrowed from Wikipedia
(define (h x)
  (lambda (y) (+ x y)))

((h 1) 2)
```

Example of a Closure

```
; Borrowed from Wikipedia
(define (h x)
  (lambda (y) (+ x y)))

((h 1) 2)
```

The reason why the above works is because the `lambda` that has the `y` parameter inherits the environment of its parent function definition.

Table of Contents

- Now that we know about environments, we can now discuss *interpreters*.
- REPL stands for:
  1. Read – parse the input expression
  2. Evaluate – evaluate the parsed expression
  3. Print – print the evaluated result
  4. Loop – wait for another expression to read, then start at Read

The core evaluation rules of Lisp's primitive functions are quite simple, yet profound since they are the bedrock of so many useful computations. Alan Kay, creator of the Smalltalk programming language, called them "the Maxwell equations of software." These will be covered in the next lecture.

Table of Contents

Your task for Project 1 is to write a Scheme metacircular interpreter; i.e., a Scheme interpreter written in Scheme. Project 1 is due October 2, 2020 at 11:59pm.