CS 450

Lecture 6: The `set!` Special Form; Programs with State

Carl D. Offner

Modeling of complex systems can be done by considernig the system as composed of objects with internal state. This state may change over time. Thus for the first time, we introduce imperative (rather than functional) programming.

1 A withdrawal processor

For instance we might have a bank account with a `balance`:

```
(define balance 100)
(set! balance 130)
```

and so on. Note:

- `set!` is a special form.
- A variable can only be `set!` if it has already been defined.
- We use the convention that the name of a procedure ends in an exclamation point if it changes the value of its argument. This explains why `set!` is spelled the way it is.

Let us define a procedure `withdraw` which acts like this:

```
==> (define balance 100)
balance
==> (withdraw 25)
75
==> (withdraw 25)
50
==> (withdraw 60)
Insufficient funds
==> (withdraw 15)
35
```

Here is the definition of `withdraw`:

```
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

Note:

- We have introduced a new special form: `begin`. `(begin <exp1> <exp2> ... <expn>)` evaluates each expression in turn and returns the value of the last one.

  Since the values of all but the last expression are discarded, the only reason for using this construct is if all the expressions except the last one have side-effects (like `set!` in this example).

- We could have avoided using `begin` if we had used `cond` instead of `if`.

2 Making the balance private

Now really, the `balance` of the bank account should not be a global variable—no one else should be able to see your balance. So we'll create a new version in which `balance` is a local variable initialized to 100:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

This works the way it ought to, and we don't have to define `balance` to start off, because it is already initialized in the procedure definition:

```
==> (new-withdraw 25)
75
==> (new-withdraw 25)
50
==> (new-withdraw 60)
Insufficient funds
==> (new-withdraw 15)
35
```

It probably seems confusing at this point just how `balance` is really managed. Why doesn't it get reinitialized on each call to `new-withdraw`? We'll see exactly how this works when we explain the environment model, later.

In the meantime, here is a question to keep in mind:

Where does the variable `balance` *live?*

For instance, we know it's not a global variable. But it's not entirely local either, because it lives (and maintains its value) between invocations of `new-withdraw`.

We won't answer that question right now, but we will soon, and the answer will be the key to understanding exactly how these procedures work.

3 Making a procedure to create "withdrawal" accounts

We can go farther and create a "withdrawal processor" creator—this way we have a procedure that creates separate bank accounts:

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

Here's how it works:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
```

`W1` and `W2` are completely independent objects!

```
==> (W1 50)
50
==> (W2 70)
30
==> (W2 40)
Insufficient funds
==> (W1 40)
10
```

Again, we will explain this in more detail later. The point here is that these two withdrawal processors work correctly and completely independently.

4 Creating real bank accounts

Finally, let's make a bank account creator, that will enable us to create a bank account into which money can be deposited (as well as withdrawn from). We can do this by modeling the bank account as a procedure that accepts messages:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT" m))))
  dispatch)
```

Here's how it works:

```
==> (define acc (make-account 100))
acc
==> ((acc 'withdraw) 50)
50
==> ((acc 'withdraw) 60)
Insufficient funds
==> ((acc 'deposit) 40)
90
==> ((acc 'withdraw) 60)
30
```

Notice:

```
(define fred-acc (make-account 100))
(define sally-acc (make-account 100))
```

yields two distinct accounts, but

```
(define fred-acc (make-account 100))
(define sally-acc fred-acc)
```

yields 1 joint account.