CS403: Programming Languages

Notes on State

Changing the state of a variable

Up until now, we have had no need to change the state of a variable. To give a variable a value, we used the *define* special form. However, for all we know, *define* just gives a name, or alias, to a value. For computing mathematical functions, *define* is sufficient. However, many real world problems require modeling the state of an object and simulating how that state changes over time. While it is possible to perform such modeling and simulation without state, the task becomes much easier if we are allowed to modify the value of a variable.

Scheme allows us to change the state of a variable with the *set!* special form. For example:

```
(define balance 100)
(set! balance (- balance 10))
```

causes *balance* to have a value of 90. Of course, we could have just redefined *balance*

```
(define balance 100)
(define balance (- balance 100))
```

but if we desire to modify the value of *balance* with a procedure, such a strategy cannot work. Here is a function which modifies *balance*:

```
(define (decrement amt)
    (set! balance (- balance amt))
    balance
    )
```

Given an initial value of 100 for *balance*, the expression

```
(decrement 10)
```

returns a value of 90, while a subsequent call:

```
(decrement 10)
```

returns a value of 80.

We can also modify local variables. Here is a message passing version of a bank account which stores the initial value as a local variable. In the example, a message-passing paradigm is used; the bank account object is a function which dispatches a message.

```
(define (newAccount balance)
    (define (account msg . args)
        (cond
            ((eq? msg 'withdraw)
                (set! balance (- balance (car args)))
                balance)
            ((eq? msg 'deposit)
                (set! balance (+ balance (car args)))
                balance)
            ((eq? msg 'balance)
                balance)
            (else
                (error "bad message: " msg))
            )
        )
    account
    )
```

Because *balance* is free with respect to the returned lambda, *balance* retains its identity throughout the existence of the lambda. Let's use *newAccount* to create and manipulate a bank account.

```
(define myAccount (newAccount 100)) ;create a new account
(myAccount 'withdraw 10)            ;should evaluate to 90
(myAccount 'deposit 30)             ;should evaluate to 120
(myAccount 'balance)                ;should evaluate to 120
```

Since *balance* is local, we can create as many different accounts as we wish and not one account will interfere with the others.

Using objects as a repository for state

We could also use Scam's object system to create a bank account:

```
(define (newAccount balance)
    (define (withdraw amt)
        (set! balance (- balance amt))
        balance
        )
    (define (deposit amt)
        (set! balance (+ balance amt))
        balance
        )
    (define (balance)
        balance
        )
    this
    )
```

The interaction analogous to the message-passing version of a bank account is:

```
(define myAccount (newAccount 100)) ;create a new account
((myAccount'withdraw) 10)           ;should evaluate to 90
((myAccount 'deposit) 30)           ;should evaluate to 120
((myAccount 'balance))              ;should evaluate to 120
```