CS 360: Programming Languages
Lecture 3: Programming with State

Geoffrey Mainland

Drexel University

Section 1

Administrivia

Administrivia

- More than 50% of the class wants to keep the Monday deadline. I will close the poll at the end of today, so vote now if you haven't already!
- Lab 1 is due this coming Monday, January 21 at midnight. It is more difficult than Homework 1, but not immensely so.
- You may work *with a partner* on **labs**. You must work *alone* on **homeworks**.
- We will have an hour at the end of Thursday's lecture (or the second half class if you're in the night class) to work on the lab. The point of this is for me to help you do the lab if you need it!

Section 2

Homework 0

Problems using git?

- Good news: no problems with git or GitHub.
- How can you tell what we will see when we collect your homework? Look at your repository on GitHub!
- You can clone the un-worked lecture examples repository with the command
  `git clone git@github.com:DrexelCS360/lectures.git`.
- After you have cloned a repository, you can update its contents with `git pull`.
- You do not need to check in generated binaries, temporary files, etc. Doing so is considered very poor form... Our supplied `.gitignore` should prevent most of this gunk from making it into your repository.

How to write main in C?

```
int main(int argc, char* argv[])
{
  ...
  return 0;
}
```

```
int main(int argc, char** argv)
{
  ...
  return 0;
}
```

```
int main(void)
{
  ...
  return 0;
}
```

See also: http://stackoverflow.com/questions/2108192/what-are-the-valid-signatures-for-cs-main-function

C program exit status

- Every C program must return an exit status from `main`.
- 0 is success, 1 is failure. Even better: use `EXIT_SUCCESS` or `EXIT_FAILURE`, which are defined in `stdlib.h`.
- If your definition of `main` was wrong or you did not correctly return `0` from `main`, you were penalized 1 point on Homework 0.

General homework advice

- I took 1 point off for incorrect “Hello, world!” output. Please read the assignment carefully. If something is unclear, please ask.
- You will have to read specifications carefully in the “real world!” I once used the incorrect version of CVS. . .
- I am not trying to trick you. If the assignment asks you to do something, that something will probably show up in the rubric.
- Please attempt every problem—even if that means just writing a specification in a comment.
- Please tell me how long you spent on each problem, even for those you didn’t finish or attempt!
- Always try to at least get the base case right even if you can’t solve the recursive case perfectly.
- These are easy points—everyone should get them.
- You should be able to solve all the homework problems using only the functions we see in lecture—you should not have to go digging through the Racket manual!
- Run the tests!

Section 3

Review

Creating lists with cons

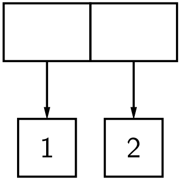
- cons creates a pair. The first element of the pair is the **car**, and the second element is the **cdr**.
- What happens if we evaluate (cons 1 2)?

  ```
  > (cons 1 2)
  '(1 . 2)
  >
  ```

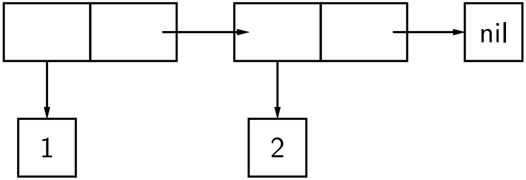
- The value (1 . 2) is called a **dotted pair**.
- What is the difference between (1 . 2) and (1 2)?

Creating lists with cons

- When we evaluate `(cons 1 2)`, the interpreter prints `(1 . 2)`, which we can visualize like this:



- When we evaluate `(list 1 2)`, the interpreter prints `(1 2)`, which we can visualize like this:



- How can we construct the value `(1 2)` using only cons, i.e., without using `list`? `(cons 1 (cons 2 '()))`

Defining lists

- A list is either **empty** or an **element paired with a list**.
- The empty list is `'()`, a.k.a, “nil” or “null.”
- How do we pair an element with a list? Using `cons`.
- There is no dot shown when displaying `(1 2)` because the final cdr is nil.
- Pairs can be used to define other data types, e.g., rational numbers.

A quiz...

- `(list* v ... tail)` is like list, but the last argument is used as the tail of the result, instead of the final element. The result is a list only if the last argument is a list. Examples:

  ```
  > (list* 1 2)
  '(1 . 2)
  > (list* 1 2 (list 3 4))
  '(1 2 3 4)
  ```

- What is the relationship between the value of the expression `(list* x y)` and the value of the expression `(cons x y)`, where `x` and `y` could themselves be arbitrary expressions?

Higher-order Functions

- In Scheme, functions **are** values.
- In particular, we can write functions that take functions as arguments and return **new** functions.
- Let's review the higher-order functions we saw last lecture and look at a few new ones.

Section 4

The substitution model of evaluation

Recall the steps we took to evaluate a Scheme expression. . .

```
(define (square x) (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f z)
  (sum-of-squares (+ z 1) (* z 2)))
```

```
(f 5)

(sum-of-squares (+ z 1) (* z 2))

(sum-of-squares (+ 5 1) (* 5 2))

(sum-of-squares 6 10)

(+ (square x) (square y))

(+ (square 6) (square 10))

(+ (* 6 6) (* 10 10))

(+ 36 100)

136
```

The substitution model of evaluation

- The values of a constant (numeric literal, character, string, etc.) is the named constant.
- The values of built-in operators are the machine instruction sequences that carry out the corresponding operations.
- The value of any other symbol is the value associated with that symbol in the environment.
- To evaluate a **compound expression**:
    - Evaluate the subexpressions.
    - Apply the procedure that is the value of the first subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).
- To apply a **primitive procedure** to arguments, the interpreter will run the machine instructions associated with the primitive. We assume the interpreter knows how to do this already.
- To apply a **compound procedure** to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

Section 5

Programming With State

State in Scheme

- Until now, we have only seen pure functions in Scheme.
- Scheme does allow impure functions—variables can in fact be modified, or assigned, in Scheme.
- We will use state to create a simple model of the world using objects—we will model bank accounts.

Withdrawing money from a bank

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

Note that evaluating `(withdraw 25)` twice yields *different* values.
Can `withdraw` be a pure function?

Implementing balance

```
(define balance 100)

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

Two new special forms:

```
(set! <name> <new-value>)
```

The `set!` special form changes <name> so that its value is the value obtained by evaluating <new-value>.

```
(begin <e1> <e2> ... <en>)
```

The `begin` special form evaluates each expression in order. The value of the special form is the value of the final subexpression, <en>.

A problem with balance...

- Unfortunately, we've used a global variable to track our account balance.
- We can solve this issue by rewriting our definition as follows:

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

Keeping state local

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

- We have used `let` to establish a **local environment** with a variable `balance`.
- Where is the variable `balance` **in scope**? That is, where are we allowed to use it?
- This lets us encapsulate the state our bank account uses. Doesn't this look a lot like objects?
- Recall the global environment from the last lecture.

Lexical vs. dynamic scoping

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

- Scheme obeys **lexical scope**: a variable is in scope in the program text of the body of the construct (e.g., let, lambda) that defines the variable.
- In contrast, in a **dynamically scoped** language, a variable is in scope for the duration of execution of the construct (e.g., let, lambda) that defines the variable.
- Major advantage of lexical scoping: we can tell what is in scope by looking at the program text. Lexical scoping makes programs easier to reason about.

Example: Lexical vs. dynamic scoping

```scheme
(define m 50)
(define n 100)

(define (hardy)
   (display (list "In Hardy, n=" n)) (newline))

(define (laurel n)
   (display (list "In Laurel, m=" m)) (newline)
   (display (list "In Laurel, n=" n)) (newline)
   (hardy))

(display (list "In main program, n=" n)) (newline)
(laurel 1)
(hardy)
```

With lexical scoping, this yields:

```
In main program, n = 100
In laurel, m = 50
In laurel, n = 1
In hardy, n = 100   ;; called from laurel
in hardy, n = 100   ;; called from main
```

Example: Lexical vs. dynamic scoping

```
(define m 50)
(define n 100)

(define (hardy)
   (display (list "In Hardy, n=" n)) (newline))

(define (laurel n)
   (display (list "In Laurel, m=" m)) (newline)
   (display (list "In Laurel, n=" n)) (newline)
   (hardy))

(display (list "In main program, n=" n)) (newline)
(laurel 1)
(hardy)
```

With dynamic scoping, this yields:

```
In main program, n = 100
In laurel, m = 50
In laurel, n = 1
In hardy, n = 1      ;; <== NOTE!!  called from laurel
in hardy, n = 100    ;; called from main
```

“Object-oriented” programming in Scheme

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

We can use `make-withdraw` to create multiple account objects, each with its own balance.

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
(W1 50)
=> 50
(W2 70)
=> 30
(W2 40)
=> "Insufficient funds"
```

A more complex account object

```scheme
(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)
```

Using the new account object

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

Section 6

Failure of the Substitution Model

Using the substitution model with a pure function

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
```

```
((make-decrementer 25) 20)
```

```
((lambda (amount) (- 25 amount)) 20)
```

```
(- 25 20)
```

```
5
```

Using the substitution model with an *impure* function

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

```
((make-simplified-withdraw 25) 20)
```

```
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
```

```
((set! balance (- 25 20)) 25)
```

- If we obeyed the substitution model, we would say that the meaning of this expression is to set `balance` to 5 and then return 25.
- Very wrong! To get the right answer, we have to somehow distinguish the first value of `balance` (before `set!`) from the second value (after `set!`).
- The problem is that the substitution model assumes variables stand for values (and never change). Introducing `set!` breaks this assumption.

The failure of the substitution model

- The substitution model only works with *pure* programs.
- We need a new model to reason about the evaluation of Scheme programs with state, e.g., Scheme with `set!`.
- Next week we will see such a model. This model will also more closely reflect what language implementations actually do under the covers.

The substitution model and equational reasoning

```
(let ((x e))
  (if #t
      0
      x))
```

- **Equational reasoning** allows us to substitute equals for equals.
- In a pure language, a binding is *almost* an equation.
- What is the value of the expression above? Does it depend on e?
- What if our language is pure and *lazy*, i.e., it doesn't evaluate expressions until they are needed?