Lecture 19: Scheme I

Marvin Zhang
07/25/2016
Announcements
This week (Interpretation), the goals are:

- To learn a new language, Scheme, in two days!
- To understand how interpreters work, using Scheme as an example
Scheme

• Scheme is a dialect of Lisp, the second-oldest language still used today

• “If you don't know Lisp, you don't know what it means for a programming language to be powerful and elegant.”
  - Richard Stallman, creator of Emacs

• “The greatest single programming language ever designed.”
  - Alan Kay, co-creator of OOP

• Lisp is known for its simple but powerful syntax, and its ridiculous number of parentheses
  • What does Lisp stand for?
Scheme Fundamentals

- Scheme primitives include numbers, Booleans, and symbols
  - More on symbols later (for now, they’re like variables)

- There are various ways to combine primitives into more complex expressions
  - Call expressions include an operator followed by zero or more operands, all surrounded by parentheses

```
(scm> (quotient (+ 8 7) 5))  ; scm> (+ (* 3
3

57)

(scm> (+ (* 2 4))

(scm> (+ 3 5))

(scm> (+ (- 10 7)

6))

57
```
Special Forms

Assignment, Symbols, Functions, and Conditionals
Assignment Statements Expressions

• **Special forms** in Scheme have special orders of evaluation

• We can bind symbols to values using **define**

  • `(define <symbol> <expression>)` binds `<symbol>` to the value that `<expression>` evaluates to

  ```scheme
scm> (define a 5)  
a
scm> (define b (+ a 4))  
b
scm> a
5
scm> b
9
```

• Everything in Scheme is an expression, meaning everything evaluates to a value

• **define** expressions evaluate to the symbol that was bound
Symbols and `quote`

- Symbols are like variables, they can be bound to values.
- However, unlike variables, they also exist on their own as their own values.
- Symbols are like strings and variables all in one.
- We can reference symbols directly, rather than the value they are bound to, using the `quote` special form.

```
scm> (define a 5)  scm> (quote a)
   a       a
scm> a              scm> 'a  ; shorthand for (quote a)
   5       a
```
Assignment Expressions

- `define` expressions evaluate to the symbol that was bound, not the value the symbol was bound to.
- The side effect of a `define` expression is to bind the symbol to the value of the expression.

```
scm> (define a 5)  
a
scm> (define b a)  
b
scm> b
5
scm> (define c (define a 3))  
c
scm> a
3
scm> c
a
```
Lambda Expressions

- **lambda** expressions evaluate to anonymous *procedures*
  - *(lambda (<parameters>) <body>)* creates a procedure as the side effect, and evaluates to the procedure itself
  - We can use the procedure directly as the operator in a call expression, e.g., `((lambda (x) (* x x)) 4)`
    - [operator] [operand]
  - More commonly, we can bind it to a symbol using an assignment, e.g., `((define square (lambda (x) (* x x)))`)
  - This is so common that we have a shorthand for this: `(define (square x) (* x x))` does the exact same thing
  - This looks like a Python **def** statement, but the procedure it creates is still anonymous!
Conditionals and Booleans

- Conditional expressions come in two types:
  - `(if <predicate> <consequent> <alternative>)` evaluates `<predicate>`, and then evaluates and returns the value of either `<consequent>` or `<alternative>`
  - We can chain conditionals together similar to Python `if–elif–else` statements using the `cond` expression

  ```scm
  scm> (cond ((= 3 4) 4)
   ((= 3 3) 0)
   (else 'hi))
  0
  ```

- Booleans expressions (`and <e1> ... <en>`), (`or <e1> ... <en>`) short-circuit just like Python Boolean expressions
- In Scheme, only `#f` (and `false`, and `False`) are false values!
Pairs and Lists

Scheme data structures
Pairs and Lists

• Disclaimer: programmers in the 1950s used confusing terms
• The pair is the basic compound value in Scheme, and is constructed using a cons expression
• car selects the first element in a pair, and cdr selects the second element

```
scm> (define x (cons 1 3))
x
scm> x
(1 . 3)
scm> (car x)
1
scm> (cdr x)
3
```
Pairs and Lists

• The only type of sequence in Scheme is the linked list, which we can create using just pairs!

• There is also shorthand for creating linked lists using the list expression

• nil represents the empty list

```
scm> (define x (cons 1 (cons 2 (cons 3 nil))))
x
scm> x ; no dots displayed for well-formed lists
 (1 2 3)
scm> (car x)    scm> (list 1 2 3) ; shorthand
 1             (1 2 3)
scm> (cdr x)    scm> '(1 2 3) ; shortest-hand
 (2 3)         (1 2 3)
```
Coding Practice

- Let’s implement a procedure \((\text{map } \text{fn } \text{lst})\), where \(\text{fn}\) is a one-element procedure and \(\text{lst}\) is a (linked) list
  - \((\text{map } \text{fn } \text{lst})\) returns a new (linked) list with \(\text{fn}\) applied to all of the elements in \(\text{lst}\)
- A good way to start these problems is to write it in Python first, using \(\text{linked lists}\) and \(\text{recursion}\)
  - Usually pretty easy to translate to Scheme afterwards
- Basic versions of Scheme don’t have iteration!

\[
\text{(define } (\text{map } \text{fn } \text{lst})
  \begin{cases}
    \text{if } (\text{null? lst}) \\
    \text{nil} \\
    (\text{cons } (\text{fn } (\text{car lst})) (\text{map } \text{fn } (\text{cdr lst})))
  \end{cases}\text{)}
\]
• We can create a tree abstraction just like in Python:

```scheme
(define (tree entry children)
  (cons entry children))

(define (entry tree) (car tree))

(define (children tree) (cdr tree))

(define (leaf? tree)
  (null? (children tree)))

(define (square-tree t)
  (tree (square (entry t))
    (if (leaf? t) nil
      (map square-tree (children t))))))
```
Summary

• We learned a new language today! Being able to quickly pick up new languages is important for good programmers.

• Scheme is a simpler language, but still very powerful.
  • Everything in Scheme is an expression.
  • All functions (called procedures) are anonymous.
  • Because the only sequence is the linked list, we will solve problems using recursion.

• “How do I master Scheme?” Go practice!