Lecture 19: Scheme I

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

Roadmap

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 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) 3
scm> (+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

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?

Special Forms

Assignment, Symbols, Functions, and Conditionals
Assignment Statements

• 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
define a 5
a
5
```

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

• `define` expressions evaluate to the symbol that was bound

```scheme
define a 5
a
```

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

```scheme
define a 5
(quote a)
5
```

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

```scheme
define a 5
(define c (define a 3))
(demo)
```

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))` creates a procedure that squares its 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!

```scheme
define a 5
define b (+ a 4)
da b
```

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

```scheme
cond
((= 3 4) 4)
((= 3 3) 0)
(else 'hi)
0
demo
```

• Boolean expressions: `(and <e1> ... <en>)` and `(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

```scheme
(scm> (define x (cons 1 3)))
```

```scheme
x
```

```scheme
(1 . 3)
```

```scheme
(scm> (car x))
1
```

```scheme
(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

```scheme
(scm> (define x (cons 1 (cons 2 (cons 3 nil)))))
```

```scheme
x
```

```scheme
; no dots displayed for well-formed lists
```

```scheme
(scm> (car x))
```

```scheme
; shorthand
```

```scheme
(scm> (cdr x))
```

```scheme
((1 2 3))
```

Coding Practice

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

```scheme
(define (map fn lst)
  (if (null? lst)
      nil
      (cons (fn (car lst)) (map fn (cdr lst))))))
```

More Coding Practice

- We can create a tree abstraction just like in Python:

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

```scheme
(define (entry tree) (car tree))
```

```scheme
(define (children tree) (cdr tree))
```

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

```scheme
(define (square-tree t)
  (tree
    (square (entry t))
    (if (leaf? t)
        (null? (children t))
        (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!