Lecture 19: Scheme I

Marvin Zhang
07/25/2016

Announcements

Roadmap

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

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Interpretation), the goals are:

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Interpretation), the goals are:
  - To learn a new language, Scheme, in two days!

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- 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

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

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

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

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

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 Fundamentals

- Scheme primitives include numbers, Booleans, and symbols

Scheme Fundamentals

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

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

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

Scheme Fundamentals (demo)

- 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

Scheme Fundamentals (demo)

- 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
```

Scheme Fundamentals (demo)

- 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))
57
```

Special Forms

Assignment, Symbols, Functions, and Conditionals

Assignment Statements

Assignment Statements

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

Assignment Statements

- *Special forms* in Scheme have special orders of evaluation
- We can bind symbols to values using **`define`**

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

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

```
scm> (define a 5)
```

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

```
scm> (define a 5)
a
```

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

```
scm> (define a 5)
a
scm> a
```

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

```
scm> (define a 5)
a
scm> a
5
```

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

```
scm> (define a 5)
a
scm> a
5
```

```
scm> (define b (+ a 4))
```

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

```
scm> (define a 5)
a
scm> a
5
```

```
scm> (define b (+ a 4))
b
```

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

```
scm> (define a 5)
a
scm> a
5
```

```
scm> (define b (+ a 4))
b
scm> b
```

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

```
scm> (define a 5)
a
scm> a
5
```

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

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

```
scm> (define a 5)
a
scm> a
5
```

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

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

```
scm> (define a 5)
a
scm> a
5
```

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

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

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

```
scm> (define a 5)
a
scm> a
5
```

```
scm> (define b (+ a 4))
b
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 and **`quote`**

- Symbols are like variables, they can be bound to values

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 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

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

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)
```

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)
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

```
scm> (define a 5)
a
scm> 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

```
scm> (define a 5)
a
scm> a
5
```

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)
a
scm> a
5
```

```
scm> (quote 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

```
scm> (define a 5)
a
scm> a
5
```

```
scm> (quote a)
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

```
scm> (define a 5)
a
scm> a
5
```

```
scm> (quote a)
a
scm> 'a  ; shorthand for (quote 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

```
scm> (define a 5)
a
scm> a
5
```

```
scm> (quote a)
a
scm> 'a  ; shorthand for (quote a)
a
```

Assignment Expressions

Assignment Expressions

- **define** expressions evaluate to the symbol that was bound, not the value the symbol was bound to

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

Assignment Expressions (demo)

- **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

Assignment Expressions (demo)

- **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

- **`lambda`** expressions evaluate to anonymous *procedures*

Lambda Expressions

- **lambda** expressions evaluate to anonymous *procedures*
  - `(lambda (<parameters>) <body>)` creates a procedure as the side effect, and evaluates to the procedure itself

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)

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

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)

$$(\underbrace{\texttt{(lambda (x) (* x x))}}_{\text{operator}}\ \underbrace{\texttt{4}}_{\text{operand}})$$

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)))`

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

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

Conditionals and Booleans

- Conditional expressions come in two types:

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>`

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

Conditionals and Booleans (demo)

- 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

Conditionals and Booleans (demo)

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

Conditionals and Booleans (demo)

- 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> (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

Conditionals and Booleans (demo)

- 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> (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

Pairs and Lists

- Disclaimer: programmers in the 1950s used confusing terms

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

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

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))
```

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
```

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
```

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)
```

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)
```

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
```

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)
```

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

Pairs and Lists

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

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

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

Pairs and Lists (demo)

- 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

Pairs and Lists (demo)

- 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)
1
scm> (cdr x)
(2 3)
```

```
scm> (list 1 2 3)  ; shorthand
(1 2 3)
scm> '(1 2 3)  ; shortest-hand
(1 2 3)
```

Coding Practice

Coding Practice

- Let’s implement a procedure `(map fn lst)`, where `fn` is a one-element procedure and `lst` is a (linked) list

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`

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*

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

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!

Coding Practice (demo)

- 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!

Coding Practice (demo)

- 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!

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

More Coding Practice

More Coding Practice

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

More Coding Practice

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

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

More Coding Practice

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

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

More Coding Practice

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

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

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

More Coding Practice

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

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

(define (entry tree) (car tree))

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

More Coding Practice

- 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)
```

More Coding Practice

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

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

(define (entry tree) (car tree))

(define (children tree) (cdr tree))

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

More Coding Practice (demo)

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

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

(define (entry tree) (car tree))

(define (children tree) (cdr tree))

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

More Coding Practice (demo)

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

```
(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

Summary

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

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

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

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

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*

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!