CS 360: Programming Languages
Lecture 2: Functional Programming in Scheme

Geoffrey Mainland

Drexel University

Section 1

An Introduction to Scheme

What is Functional Programming?

- Functions in a program are mathematical functions—for a given input, there is a unique output.
- No assignment to variables.
- Therefore... no loops.
- Typically, functions are first-class.

Why Functional Programming?

- FP encourages you to think *compositionally*.
- FP encourages you to think about *abstraction*.
- FP discourages the use of mutable state—helps with, e.g., parallelism.
- Even if you program in C++, these are useful skills.
- I hope you will one day find yourself thinking, "If I were using a functional language, I could do things this way. . . "
- Many idea from FP are making their way into "mainstream" languages, e.g., C++ and Swift.

Getting Started with Scheme

- We will use The Racket variant of Scheme, available as `racket` or `drracket` on `tux`. If you are using the course virtual machine, use the DrRacket IDE.
- See the Racket guide linked to from the course home page, especially for tips about debugging with the IDE.
- There are links to other Scheme resources on the course web page.

Elements of Scheme

A good programming language should provide ways to combine simple ideas to form more complex ideas. There are three mechanism for accomplishing this:

- **Primitive expressions** are the simplest entities in the language.
- **Means of combination** allow us to build complex elements from simpler elements.
- **Means of abstraction** allow us to take complex elements and name and manipulate them as units.

Primitive Expressions in Scheme

Scheme includes the primitive expressions we expect of any language.

Expression	What it represents
`42`	an integer value
`"hello"`	a string
`#t`	Boolean truth
`#\a`	the character 'a'
`+`	the symbol +
`x`	the symbol x

Combining Expressions in Scheme

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

- Expressions are combined in Scheme by enclosing a list of expressions in parentheses.
- A parenthesized list of expressions represents a **procedure application**.
- The first element in the list is called the **operator**, and the other elements are called **operands**.
- We evaluate this **compound expression** by applying the procedure specified by the operator to the arguments that are the values of the operands.
- The convention of placing the operator to the left of the operands is known as **prefix notation**.

Variables in Scheme

```scheme
(define pi 3.14159)
```

- A good programming language provides a way to use names to refer to objects.
- A name, also known as a **symbol**, identifies a **variable** whose **value** is the object.
- In Scheme, we name things using `define`.
- `define` is Scheme's simplest form of abstraction—it allows us to use simple names to refer to the results of complex operations.
- Internally, the interpreter must keep track of the values associated with any variables.
- This mapping is called the **environment** (or more precisely the **global environment**).

Scheme: Evaluation

- The values of a constant (numeric literal, character, string, etc.) is the named constant.
- The value of a built-in symbol, like +, is the sequence of instructions needed to carry out the corresponding operation.
- 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).

Scheme: Special Forms

```scheme
(define pi 3.14159)
```

- Note that the previous rules do not handle evaluating `define`.
- That is, the above list of expressions does not apply `define` to two arguments.
- Exceptions to the general evaluation rule are called **special forms**. We will see other special forms later.

The identity function in Scheme

How can we write the identity function in Scheme?

```
(lambda (x) x)
```

- A lot like the lambda calculus...
- Syntax is `(lambda (<formal parameters>) <body>)`.
- How is the above identity function evaluated?
- How is `lambda` different from the lambda-calculus λ?

Defining Functions in Scheme

How can we bind the identity function to a variable (symbol) `id`?

```
(define id (lambda (x) x))
```

Since defining functions this way is so common, there is a shorthand:

```
(define (id x) x)
```

Syntax is `(define (<name> <formal parameters>) <body>)`.
This is also called a **compound procedure**.

A model for the evaluation of Scheme expressions

```scheme
(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)))
```

- How can we think about the way in which `(f 5)` is evaluated?
- To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

```scheme
(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
```

A model for the evaluation of Scheme expressions

- This is known as the **substitution model** for the evaluation of scheme.
- This is only a model—it isn't meant to precisely describe the inner workings of the interpreter.
- Where have we seen this model before?

Alternative evaluation methods

- To evaluate a compound expression, we evaluate the operator and operands and then apply the resulting procedure to the arguments. This is called **applicative order** evaluation.
- Alternative: don't evaluate arguments until they are needed. This is called **normal order** evaluation.

Normal order evaluation

```
(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))
```

```
(+ (square (+ 5 1)) (square (* 5 2)))
```

```
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
```

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

```
(+ 36 100)
```

```
136
```

Order of evaluation

- Applicative order, also called **call-by-value**, evaluates arguments before calling a function.
- Normal order, also called **call-by-name**, evaluates arguments only when they are actually needed. Note that normal order and call-by-name are not actually quite the same thing, but we won't worry about the distinction at the moment.
- Can you think of pros and cons of these two evaluation strategies?
- **Call-by-need memoizes** results to avoid recomputing them.
- How would you change the evaluation rules that we saw in lecture 1 for the lambda calculus so that normal order evaluation was used?

Conditional Expressions in Scheme

```
(define (abs x)
  (cond
   ((> x 0) x)
   ((= x 0) 0)
   ((< x 0) (- x))))
```

General form:

```
(cond (p1 e1)
      (p2 e2)
      ...
      (pn en))
```

Conditional Expressions in Scheme cont'd

```
(define (abs x)
  (cond
   ((> x 0) x)
   ((= x 0) 0)
   ((< x 0) (- x))))
```

Could be written like this:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

General form:

```
(if <predicate>
    <consequent>
    <alternative>)
```

Defining variables with `let`

General form:

```
(let ((v1 e1)
      (v2 e2)
      ...
      (vn en))
  body)
```

Data types in Scheme

- In addition to the base types (numbers, characters, strings, etc.), Scheme has **pairs**.
- Pairs are created using the `cons` function.
- For historical reasons, the first element of the pair is called the `car`, and the second element is called the `cdr`.
- `car` and `cdr` are also the names of Scheme functions that extract the first and second element of a pair, respectively.

Lists in Scheme

- The empty list, often called “nil” or “null,” is primitive in Scheme. It is written as `'()` or as `null`. In Racket, `null` is preferred.
- How can we build (non-empty) lists of integers from integers, pairs, and nil?
- The function `list` builds a list from its arguments.

Scheme: Some Special Forms

- What special forms have we already seen?
- Do you think the Scheme functions and and or are special forms?
- quote is a special form that simply returns its argument **without evaluating it**

  ```
  (quote (1 "hello")) => (1 "hello")
  '(1 "hello") => (1 "hello")
  ```

- Could we implement quote using list?

  ```
  (list 1 "hello") => ...
  ```

Scheme predicates

- Scheme **predicates** are functions that perform a test and return true or false. By convention, they end with ?.
- Some Scheme predicates you may need: `pair?`, `list?`, `null?`, `integer?`.
- See the course home page for links to the Racket and Scheme language references.

Programming with lists

Let's work some examples...

- Some of our examples used *deep recursion*—they recursed into nested lists.
- Most used *shallow recursion*—they did not recurse into nested lists.

Higher-order Functions

- **Higher-order** functions are functions that take other functions as an argument.
- Lambdas and higher-order functions are a feature once only seen in functional languages, but they have made their way into other languages...

```
auto identity = [](auto x) {
  return x;
};
```

- One of the simplest examples is `map`. Let's see an example...
- Others you may find useful are `filter` and `reduce`. Let's see more examples...