How Scheme Evaluates Expressions (version 2)

Summary: We extend our understanding of the algorithm and structures that the Scheme interpreter users to evaluate Scheme expressions. In particular, we see how the introduction of user-defined procedures affects the operation of the interpreter.

Introduction

As you may recall, we left our original exploration of the way Scheme evaluates with a high-level overview of an algorithm the interpreter might use and a data structure that it relies on. The data structure is a *dictionary* that maps names to values. It keeps track of the values we've defined. The algorithm was something like the following:

```
Look at the next non-space character
If the next non-space character is an open parenthesis
  Look at the next thing after the open parenthesis.
  If the next thing is the keyword define
    Read the next thing (a name to define)
    Read the next thing (an expression)
    Evaluate the expression, giving a value
    Add the [name:value] entry to the dictionary
  Otherwise the next thing must be a function
    Read and evaluate each argument
    Apply the function to the evaluated arguments
Otherwise, see if the next non-space character is a digit (or + or -)
  Read all the parts of a number
  The number is the result
Otherwise, see if the next non-space character is a letter
  Read everything up to the next space (or close paren).
  Look up the thing just read in the dictionary.
  If it's not there, crash and burn
  Otherwise, the result is the value found in the dictionary
Otherwise, we've encountered an unknown kind of value
```

We've just introduced user-defined procedures, which have the following form.

```
(define procedure-name
  (lambda (parameter-list)
    expression-1
    ...
    expression-n))
```

What effects does this introduction have? Surprisingly many. We need to consider what happens in a procedure definition and what happens when we apply a procedure.

Interpreting Procedure Definitions

Suppose we have the following procedure definition.

```
(define square
  (lambda (x)
    (* x x)))
```

What does the interpreter do with this? Well, we know that when it sees an open paren, it looks at the next thing. If the next thing is the keyword `define`, it assumes it has a definition of the form `(define` *`name expression`*`)`, evaluates the expression and then associates it with *name*. So, how do we evaluate the "expression"? Here's an interesting trick: In Scheme, you *don't* evaluate lambda expressions; you leave them as is.

Hence, the effect of the previous definition is to extend the dictionary as follows

Name	Value
square	(lambda (x) (* x x))

To accommodate this approach, we need to change the interpretation algorithm a bit. In particular, if the first thing after an open parenthesis is the keyword `lambda`, we simply return the whole thing.

```
If the next non-space character is an open parenthesis
  Look at the next thing.
  If the next thing is the keyword define
    ...
  Otherwise, if the next thing is the keyword lambda
    Read until the corresponding close parenthesis.
    The whole expression is the result.
  Otherwise, the thing after the open paren is a function
    ...
```

But if the lambda is never evaluated, what good is it? It turns out the lambda *is* evaluated, just not until you use the defined procedure.

Applying Lambda

So, what happens when we apply a user defined procedure, which we've just learned is stored as an unevaluated lambda expression? Once again, we evaluate all of the arguments. It's then time to apply the procedure. In effect, Scheme begins by updating the dictionary so that each formal parameter (the things that start the lambda expression) gets associated with the corresponding argument. That is, in applying `square` to `32`, Scheme updates the dictionary to associate `32` with *`x`*. Similarly, in applying `(lambda (first second) (- (max first second) (min first second)))` to `7` and `11`, the interpreter associates the name `first` with the value `7` and the name `second` with the value `11`.

Once Scheme updates the dictionary, it evaluates the expressions in the body of the procedure, using the updated dictionary. When it reaches the end of the body, it uses the value of the last expression as the value of the procedure application.

Is the interpreter done once it's computed the value of the last expression? No. It also needs to clean up the dictionary by removing any definitions that it added at the beginning.

The Evaluation Algorithm, Revisited

```
1. Look at the next non-space character
2. If the next non-space character is an open parenthesis
  2.1 Look at the next thing after the open paren.
  2.2 If the next thing is the keyword define
    2.2.1. Read the next thing after the keyword (a name to define)
    2.2.2. Read the next thing after the name (an expression)
    2.2.3. Evaluate the expression, giving a value
    2.2.4. Add the [name:value] entry to the dictionary
  2.3. Otherwise, if the next thing is the keyword lambda
    2.3.1. Read until the corresponding close parenthesis.
    2.3.2. The whole expression is the result.
  2.4. Otherwise the next thing must be a function
    2.4.1. Look up the function in the dictionary
    2.4.2. Read and evaluate each argument
    2.4.3. If the function is a built-in function, apply it directly
    2.4.4. Otherwise, the function is a user-defined function of the
    form (lambda (param1 ... paramm) body1 ... bodyn)
      2.4.4.1. Make sure that the number of arguments equals the number of
        parameters (m).  If not, report an error.
      2.4.4.2. For each i, from 1 to m,
        2.4.4.2.1 Update the dictionary to associate parameter i with
         argument i,
      2.4.4.3. Evaluate expressions body1 ... bodyn.
      2.4.4.4. Undo the updates made to the dictionary.
      2.4.4.5. The value is the value of the bodyn.
3. Otherwise, see if the next non-space character is a digit (or + or -)
  3.1. Read all the parts of a number
  3.2. The number is the result
4. Otherwise, see if the next non-space character is a double quotation mark
   (").
   4.1. Read until the corresponding double quotation mark is found.
   4.2. The result is a string.
5. Otherwise, see if the next non-space character is a letter
  5.1. Read everything up to the next space (or close paren).
  5.2. Look up the thing just read in the dictionary.
  5.3. If it's not there, crash and burn
  5.4. Otherwise, the result is the value found in the dictionary
6. Otherwise, we've encountered an unknown kind of value
```

An Example

Let's consider an example. Suppose we have three things to evaluate: A definition of the procedure `square`, a call to that procedure on the value 5, and a call to that procedure on the sum of 3 and 4.

```
> (define square (lambda (x (* x x))))
> (square 5)
```

We'll begin with the `define`. What happens?

- 2. The next non-space character is an open parenthesis.
- 2.2. The next thing after the open parenthesis is the keyword `define`.
- 2.2.1. The name being defined is `square`.
- 2.2.2. The expression being used is `(lambda (x) (* x x))`.
- 2.2.3. We evaluate the expression.
  - 2. The next non-space character is an open parenthesis.
  - 2.3. The next thing after the open parenthesis is the keyword `lambda`.
  - 2.3.1. We read until the closing parenthesis.
  - 2.3.2. The result is `(lambda (x) (* x x))`.
- 2.2.4. We add [`square`:`(lambda (x) (* x x))`] to the dictionary.
- We're done evaluating the first expression.

As things stand, the dictionary now contains only one entry.

Name	Value
`square`	`(lambda (x) (* x x))`

Now, we're ready to evaluate the expression `(square 5)`.

- 2. The next non-space character is an open parenthesis.
- 2.4. The next things after the open parenthesis must be a function. It's name is `square`
- 2.4.1. Look up the function in the dictionary. It's `(lambda (x) (* x x))`>
- 2.4.2. Read and evaluate each argument. There's only one.
  - 3. The next character is a digit (`5`).
  - 3.1. Read all the parts of the number. The `5` is it.
  - 3.2. 5 is the result.

  So the first (and only) argument has the value 5.
- 2.4.4. We have a user-defined function of the form `(lambda (x) (* x x))`
  - There is one parameter, `x`.
  - There is one body expression, `(* x x)`.
- 2.4.4.1. The number of argument (just one) equals the number of parameters (just one).
- 2.4.4.2. We will just use parameter 1 and argument 1.
  - 2.4.4.2.1. Update the dictionary to associate `x` with `5`.

  The dictionary now has the form

Name	Value
`square`	`(lambda (x) (* x x))`
x	`5`

- 2.4.4.3. We evaluate the body, `(* x x)`.
  - 2. The next character is an open parenthesis.
  - 2.4. The thing after the open parenthesis is a function.
  - 2.4.1. We look the function up and discover that it's a built-in function.
  - 2.4.2. We evaluate each argument.
    - The first argument is `x`
      - 5. The next non-space character is a letter.
      - 5.1. We read until we're beyond the identifier. It's just `x`.
      - 5.2. We look up the `x` in the dictionary and find 5.
      - 5.4. The result is therefore 5.

      The first argument is 5.
    - The second argument is `x`
      - 5. The next non-space character is a letter.
      - 5.1. We read until we're beyond the identifier. It's just `x`.
      - 5.2. We look up the `x` in the dictionary and find 5.
      - 5.4. The result is therefore 5.

    So, the arguments are 5 and 5
  - 2.4.3. We apply the multiplication operation to 5 and 5. The result is 25.

  The body has the value 25.
- 2.4.4.4. We undo the updates to the dictionary. It now has the form

Name	Value
`square`	`(lambda (x) (* x x))`

- 2.4.4.5. The value is the 25 computed in step 2.4.4.3.

That's a lot of steps to compute the 25. Fortunately, computers are fast!

Disclaimer! We've Lied

In the discussion of applying lambda expressions to values, we suggested that the Scheme interpreter updates the dictionary and then evaluates the body of the expression. However, Scheme does a bit more trickery to limit access to the parameters of a procedure. Hence, the claim that the interpreter simply updates the dictionary is a simplification, albeit a useful one. For most of the code you write in this class (and elsewhere), the simplification is acceptable. However, there are some cases in which the simplification breaks down. We'll mention them when they occur.

For those who like esoteric terminology, the model we've given you is *dynamically scoped*. However, the Scheme language is actually *statically scoped*. For those of you who don't like esoteric terminology, ignore the preceding two sentences. (We thought about telling you to ignore the whole paragraph, but that means you would have to ignore the instruction telling you to ignore the paragraph, which could lead to some confusion.)

This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.