By Joshua Cantrell
jjc@cory.berkeley.edu

Evaluating Scheme Expressions

What is a Scheme expression?

A Scheme expression is simply something that is made using Scheme's forms[1]. These expressions can then be evaluated to return a useful value. The skill of evaluating a Scheme expression without an interpreter is extremely valuable. Without that ability, writing programs is extremely difficult because you cannot tell what will produce the desired results.

Some Simple Expressions

The simplest expressions you should become an expert at evaluating are the non-list forms. Self-evaluating expressions, such as *numbers* and *boolean*, simply evaluate to their own values. A non-evaluated self-evaluating expression is simply the sameas the evaluated expression's value. The symbol ⇒ means "evaluates to."

- `#T` ⇒ `#T` (the boolean value *#T*)
- `#F` ⇒ `#F` (the boolean value *#F*)
- `-13` ⇒ `-13` (the numeric value *-13*)
- `4/5` ⇒ `.8` or `4/5` (the numeric value *0.8* or rational numeric value *4/5ths*[2])
- `+34.23` ⇒ `34.23` (the numeric value *+34.23*)

The other common expression you have to know about is the *symbol*. Symbols, which act as identifiers, are normally bound to some other value. When you evaluate them, they are replaced with the value to which they are bound. If they are not bound to a value when evaluated, an error results. For the following examples, assume the symbol **x** is bound to the number **5**, the symbol **y** is bound to the *unevaluated symbol* **gray**, and the symbol **fn** is bound to the procedure **`#<(x) (* x x)>`**[3].

- `x` ⇒ `5` (the number 5)
- `y` ⇒ `gray` (the unevaluated symbol *gray*)
- `z` ⇒ **`ERROR`** (this never truly finishes being evaluated because it results in an error)
- `fn` ⇒ `#<(x) (* x x)>` (the procedure *#<(x) (* x x)>*)

---

[1] Look at my document on *Normal Forms & Special Forms* for an explanation on what forms are, and examples of some forms in Scheme.

[2] Whether it stays rational or not is dependant on the Scheme interpreter's implementation.

[3] I choose to represent procedures as **`#<(`P_1 P_2 `...` P_n`)` E_1 E_2 `...` E_m`>`**, where P_1 through P_n are the formal parameter symbols (aka, names or identifiers), and E_1 through E_n are the expressions that make up the body of the procedure. At this point you probably only know about procedures with one expression in their body. This is fine since that is always a possibility. You can have zero or more formal parameters, so $n \geq 0$, and you must have at least one expression in the body of a procedure, so $m > 0$. A procedure without formal parameters represented as **`#<()`** E_1 E_2 `...` E_m**`>`**.

By Joshua Cantrell
jjc@cory.berkeley.edu

Another fairly simple expression to evaluate is a quoted expression. A quoted expression is one that is not evaluated. If it were not quoted, it would be evaluated normally. There are some funny cases where a quote's result is specific to a particular interpreter, but I'll just cover the normal cases.

- `#T ⇒ #T (the boolean value *#T*)
- `+34.23 ⇒ +34.23 (the numeric value *+34.23*)
- `symbol ⇒ symbol (the unevaluated symbol value *symbol*)
- `(fn 23 0) ⇒(fn 23 0) (the unevaluated list expression *(fn 23 0)*)
- `((fn 23) 0) ⇒((fn 23) 0) (the unevaluated list expression *((fn 23) 0)*)
- `(fn1 (fn2 23) 0) ⇒(fn1 (fn2 23) 0)
  (the unevaluated list expression *(fn1 (fn2 23) 0)*)

List Expressions

Without list expressions, Scheme wouldn't be a LISPish language. This is because LISP stands for *LISt Processing*. It shouldn't be a surprise to you that understanding how to evaluate these list expressions is crucial to being able to evaluate the majority of useful Scheme expressions.

The most basic and important of these list expressions is the *procedure application list*[4]. This is the normal form of a list expression in Scheme. Anything else is considered to be a *special form*. The procedure application list is evaluated by first evaluating all of the expressions (items) inside the list. Next, given that the first expression evaluated to a procedure, the procedure is applied to the remaining elements in the list (the *actual parameters*). If the first expression did not evaluate to a procedure, an error occurs. If the procedure has more or less *formal parameters* than the number of provided *actual parameters*, an error occurs.

For the following examples the symbol **x** is bound to the number **5**, the symbol **y** is bound to the unevaluated symbol **gray**, and the symbol **fn** is bound to the procedure **#<(x) (* x x)>**. The newly evaluated elements will be in bold and the unchanged elements in normal type face. Any pre-evaluated value is underlined.

1. **(* 4 2)** - the expression to evaluate
2. (**#<*>** 4 2) - the first expression in the list, the symbol *, is evaluated[5]
3. (#<*> **4** 2) - the second expression in the list, the number *4*, is evaluated
4. (#<*> 4 **2**) - the third expression in the list, the number *2*, is evaluated
5. **8** - with all expressions in the list evaluated, the procedure is applied to the actual parameters 4 and 2, resulting in the list being replaced by the number (value) 8.[6]

---

[4] I made this term up not knowing what the standard term would be.

[5] When I don't know what the argument list and body of a procedure look like, I'll take the liberty to represent the procedure as #<*fn*> where *fn* is the symbol to which the procedure is bound.

[6] Not knowing what the body of the body of the multiplication procedure looks like, we must jump from the application of the procedure to the answer. For procedures where we know what the bodies look like, we do a substitution of the list with the body of the procedure.

By Joshua Cantrell
jjc@cory.berkeley.edu

---

1. **(y x)** - the expression to evaluate
2. (**gray** x) - the first expression in the list, *y*, is evaluated
3. (gray **5**) - the second expression in the list, *x*, is evaluated
4. **ERROR** - evaluation is terminated because the symbol *gray* is not a procedure.

---

1. **((#<() fn>) x)** - the expression to evaluate
2. (**fn** x) - the first expression in the list, *(#<() fn>)*, is a list expression, so we evaluate its first expression. Its elements have already been evaluated, so we apply the procedure to no actual parameters and replace the list expression with the body of the procedure.
3. (**#<(x) (* x x)>** x) - the first expression in the list, *fn*, is evaluated
4. (#<(x) (* x x)> **5**) - the second expression in the list, *x*, is evaluated
5. **(* 5 5)** - with all expressions in the list evaluated, the procedure is applied to the actual parameter 5, and the list is substituted with the body of the procedure with all the *x* symbols in the procedure's body replaced with the value 5.[7]
6. (**#<*>** 5 5) - the first expression in the list, *, is evaluated
7. **25** - with all expressions in the list evaluated, the procedure is applied to the actual parameters 5 and 5.

***Define* Special Form**

One of the more important special forms needed to be evaluated is the *define* special form. This special form allows us to bind symbols to values. The result of binding a symbol to a value is mostly invisible to us at this point[8]. We just need to keep track of the *scope*[9] of the symbol (identifier) which is bound to the value. The returned result of evaluating a *define* is unspecified (in other words, it can be "anything"). I'll go over define when I cover the *lambda* special form.

---

[7] I'm using the substitution method because that's what we know at this point. This is *not* how Scheme actually evaluates expressions. It's an accurate representation for functional procedures only.

[8] When we learn about environment diagrams, it'll be more clear what binding a variable entails.

[9] This is done by first looking in the procedure where the symbol is being evaluated, if the symbol wasn't bound in that procedure, we look in the procedure that the currently being evaluated procedure is defined, if not there, we continue up through the hierarchy of procedures until we reach the global environment (where you type in commands to be interpreted). This is known as *lexical scope* or alternatively *static scope.*

By Joshua Cantrell
jjc@cory.berkeley.edu

Lambda Special Form

The *lambda* special form is the constructor for producing data of type procedure. As another one of the more powerful special forms in Scheme, this should be studied carefully. Notice that a procedure is a *first-class* data type, so it can be used as an argument, be returned by a procedure, be bound to a symbol, and be stored in a list. These are the same properties that boolean, numerical, and unevaluated symbolic data have. Below are examples of how to evaluate lambda.

1. **`(lambda () 5)`** - the expression to evaluate
2. **`#<() 5>`** - the result of evaluating the lambda expression, a procedure with no arguments and a body of the expression *5*. Note that the body has **not** been evaluated yet.

1. **`((lambda (x) (* 5 2)) 24)`** - the expression to evaluate
2. (**`#<(x) (* 5 2)>`** `24`) - the result of evaluating the first expression in the list expression. This was a lambda of one formal parameter, *x*, and body *`(* 5 2)`*
3. (`#<(x) (* 5 2)>` **`24`**) - the result of evaluating the second expression in the list expression
4. **`(* 5 2)`** - the result of applying the procedure to the actual parameter[10]
5. (**`#<*>`** `5 2`) - the result of evaluating the first expression in the list expression.[11]
6. (`#<*>` **`5`** `2`) - the result of evaluating the second expression in the list expression.
7. (`#<*> 5` **`2`**) - the result of evaluating the third expression in the list expression.
8. **`10`** - the result of applying the multiply procedure

1. **`((lambda (x y) (lambda () (list x y))) 5 'gray)`**
2. (**`#<(x y) (lambda () (list x y))>`** `5 'gray`)[12]
3. (`#<(x y) (lambda () (list x y))>` **`5`** `'gray`)
4. (`#<(x y) (lambda () (list x y))> 5` **`gray`**)
5. **`(lambda () (list 5 gray))`**[13]
6. **`#<() (list 5 gray)>`**[14]

---

[10] Note that we don't have to actually use the formal parameter in the body of the procedure.
[11] Note that we didn't start evaluating the body of the procedure until it was applied in a *procedure application list.*
[12] Once again notice that the body of the procedure remains unevaluated until application.
[13] Here we just substituted the list being evaluated with the body of the applied procedure.
[14] Note that procedures can return procedures!

By Joshua Cantrell
jjc@cory.berkeley.edu

Big Procedure Example

Sometimes while evaluating scheme expressions can get gruesome (especially using the substitution method). As an example which may stimulate your mind fully, I've chosen one of these tedious examples. For this example, assume that the expression

```
(define (t f) (lambda (x) (f (f (f x)))))
```

was evaluated previously. This will bind the symbol *t* to

```
#<(f) (lambda (x) (f (f (f x))))>
```

Also assume that a procedure is bound to the symbol *1+* that has one argument and adds one to it. I'll represent this procedure by #<1+>.

1. **(((t t) 1+) 0)**
2. (((**#<(f) (lambda (x) (f (f (f x))))>** t) 1+) 0)
3. (((#<(f) (lambda (x) (f (f (f x))))>
   #<(f) (lambda (x) (f (f (f x))))>) 1+ 0)
4. ((**(lambda (x) (*#<(f) (lambda (x) (f (f (f x))))>*
   (#<(f) (lambda (x) (f (f (f x))))>
   (#<(f) (lambda (x) (f (f (f x))))> x))))**
   1+) 0)[15]
5. ((**#<(x) (#<(f) (lambda (x) (f (f (f x))))>**
   (#<(f) (lambda (x) (f (f (f x))))>
   (#<(f) (lambda (x) (f (f (f x))))> x))**>** 1+) 0)
6. ((#<(x) (#<(f) (lambda (x) (f (f (f x))))>
   (#<(f) (lambda (x) (f (f (f x))))>
   (#<(f) (lambda (x) (f (f (f x))))> x))> **#<1+>**) 0)
7. (**(#<(f) (lambda (x) (f (f (f x))))>**
   (#<(f) (lambda (x) (f (f (f x))))>
   (#<(f) (lambda (x) (f (f (f x))))> ***#<1+>*****))** 0)
8. ((#<(f) (lambda (x) (f (f (f x))))>
   (#<(f) (lambda (x) (f (f (f x))))>
   (lambda (x) (*#<1+>* (*#<1+>* (*#<1+>* x))))) 0)
9. ((#<(f) (lambda (x) (f (f (f x))))>
   (#<(f) (lambda (x) (f (f (f x))))>
   #<(x) (#<1+> (#<1+> (#<1+> x)))>) 0)
10. ((#<(f) (lambda (x) (f (f (f x))))>
    **(lambda (x) (*#<(x) (#<1+> (#<1+> (#<1+> x)))>*
    (#<(x) (#<1+> (#<1+> (#<1+> x)))>
    (#<(x) (#<1+> (#<1+> (#<1+> x)))> x))))**) 0)
11. ((#<(f) (lambda (x) (f (f (f x))))>
    #<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
    (#<(x) (#<1+> (#<1+> (#<1+> x)))>
    (#<(x) (#<1+> (#<1+> (#<1+> x)))> x))**>**) 0)

[15] I've italicized what was substituted in for the formal parameter *f* so it would be easier to see.

By Joshua Cantrell
jjc@cory.berkeley.edu

12. ((**lambda (x) (*#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))>
(#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))>
(#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))>* x)))) 0)
13. (#<(x) (#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))>
(#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))>
(#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))> x)))> 0)
14. (#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))>
(#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))>
(#<(x) (#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
(#<(x) (#<1+> (#<1+> (#<1+> x)))>
x)))> 0)))
15. **27**[16]

[16]I'm skipping the rest because you should get the idea by now. All you have to do is apply the rest of the procedures in the correct order to get the correct answer.