By Joshua Cantrell
jjc@cory.berkeley.edu

Higher Order Procedures and Lexical Scope

Procedures As Data

In my document *Evaluating Scheme Expressions*, it's shown that the *lambda* special form evaluates to data that's the procedure type. It's important to notice that this procedure data type is a *first-class* data type. In fact, most Scheme data types are first-class! Numbers, boolean, and symbols are also first-class data types. This means that numbers and procedures share some common features:

- A symbol can be bound to it.
- It can be used as an actual parameter[1] in a procedure call.
- Calling procedures can evaluate to it.[2]
- It can be used in compound data[3].

Binding Symbols to Procedures

Already we have been binding symbols to procedures extensively. Up until this point, every time we define a procedure, we bind a symbol to it[4]. This is because we haven't become proficient at using the *lambda* special form to create procedures. For example, we want to define a procedure that takes the square of a number:

1. `(define (square x) (* x x))`
2. `(define square (lambda (x) (* x x)))`
3. `(define square #<(x) (* x x)>)`
4. The symbol *square* is now bound to the procedure, `#<(x) (* x x)>`.

We don't hesitate in immediately binding the resultant procedure to the symbol *square*. Partially because what if we didn't bind the procedure to a symbol at all?

1. `(lambda (x) (* x x))`
2. `#<(x) (* x x)>`

All we did was make a procedure and have the interpreter display the result! What a waste! This is the same as typing in the number *5* just to have the interpreter reecho it back at you! That's why we usually bind some symbol, like *x*, to the number *5*, and bind some symbol, like *square*, to the procedure `#<(x) (* x x)>`. The values aren't as useful echoing on the display as they are being bound to symbol for future use.

---

[1] The arguments to the procedure which will be bound to the formal parameter symbols.
[2] In other words, the return value of a procedure can be one.
[3] Compound data is data made up of many basic types of data (like a compound molecule). Procedures can be used to create compound data, and later we'll learn about the *pair* which can also be used to create compound data.
[4] I'm talking to the general student populous in the course at the beginning of the 2nd week of instruction.

By Joshua Cantrell
jjc@cory.berkeley.edu

Not Binding Symbols to Procedure

You should recall that you don't *have to* bind a symbol to a number 5 to use it. Neither do you *have to* bind a symbol to a procedure to use it. The following is an example:

```
1. (* 5 5)
2. (#<*> 5 5)
3. 25
```

The number *5* was used in the above expression, yet it wasn't bound to a symbol. The primitive multiply procedure was bound to a symbol, but we can't *make* primitive procedures, those are built-in. We can do the same with procedures:

```
1. ((lambda (x) (* x x)) 5)
2. (#<(x) (* x x)> 5)
3. (* 5 5)
4. (#<*> 5 5)
5. 25
```

Look mom! No symbols![5]

Procedures as Actual Parameters

Symbols can be bound to procedures and numbers... Procedures and numbers can be used as actual parameters! By looking at the example above, you can see that we did indeed pass a numerical value as the actual parameter in a procedure call. But what about procedures? Here's a really simple example[6]:

```
1. ((lambda (x) x) (lambda (x) (* x x)))
2. (#<(x) x> #<(x) (* x x)>)
3. #<(x) (* x x)>
```

It didn't do much, but we still passed the procedure as an actual parameter. Neat! A more useful example might be:

```
1. ((lambda (op) (op 5 5)) *)
2. (#<(op) (op 5 5)> #<*>)
3. (#<*> 5 5)
4. (#<*> 5 5)
5. 25
```

You should start thinking about all the possible applications of this extremely useful discovery!

---

[5] This is relating the phrase, "Look mom! No hands!" which is used when children become good at riding bicycles and can finally remain balanced without using the handlebars. Like the child who no longer needs handlebars to ride a bike, we no longer need symbols to use a procedure!
[6] Don't worry about having the procedure call resulting in a procedure, that'll be covered next.

By Joshua Cantrell
jjc@cory.berkeley.edu

A Procedure as a Result of Calling a Procedure

Most of our procedures up to this point have been returning useful values. Our square procedure returns a numerical value. What about a procedure returning a procedure value? Couldn't that be useful? Here's a trivial example:

```
1. ((lambda () (lambda (x) (* x x))))
2. (#<() (lambda (x) (* x x))>)
3. (lambda (x) (* x x))
4. #<(x) (* x x)>
```

This certainly isn't the most exciting use of calling procedures to get procedures. But what about having procedures build more complex procedures? We pass numbers in and get different numbers out based on the previous numbers. Why can't we pass procedures in and get different procedures out based on the previous procedures? Here's an example that takes two procedures and returns a single procedure based on both of them:

```
1. ((lambda (p1 p2) (lambda (x) (p2 (p1 x)))) truncate integer?)
2. (#<(p1 p2) (lambda (x) (p2 (p1 x)))> #<truncate> #<integer?>)
3. (lambda (x) (#<integer?> (#<truncate> x)))
4. #<(x) (#<integer?> (#<truncate> x))>
```

Hmmm... Now this procedure can do something useful. First it applies the first procedure to a given value and then it applies the second procedure to the result of the first application. Let's see an example:

```
1. (#<(x) (#<integer?> (#<truncate> x))> 3.14)
2. (#<(x) (#<integer?> (#<truncate> x))> 3.14)
3. (#<integer?> (#<truncate> 3.14))
4. (#<integer?> 3.0)
5. #T
```

By Joshua Cantrell
jjc@cory.berkeley.edu

A Procedure in Compound Data

A procedure is a data type, right? Why not form compound data out of procedures? How about the following procedure, won't it store data in a procedure?

```
(define (contain a b)
  (lambda (x)
    (if (= x 0) a
         b)))
```

Hmmm... Let's find out:

```
1. (contain 5 *)
2. (#<(a b) (lambda (x) (if (= x 0) a b))> 5 #<*>)
3. (lambda (x) (if (= x 0) 5 #<*>))
4. #<(x) (if (= x 0) 5 #<*>)>
```

Look! The number *5*, and the procedure *#<*>* were *stored* in the procedure! Now we just need to know how to get them out... What if we called the procedure with *0* as its actual parameter?

```
1. (#<(x) (if (= x 0) 5 #<*>)> 0)
2. (#<(x) (if (= x 0) 5 #<*>)> 0)
3. (if (= 0 0) 5 #<*>)
4. (if (#<=> 0 0) 5 #<*>)
5. (if #T 5 #<*>)
6. 5
```

EUREKA! It returned the stored *5*! What if we called the procedure with something other than the number *0*?

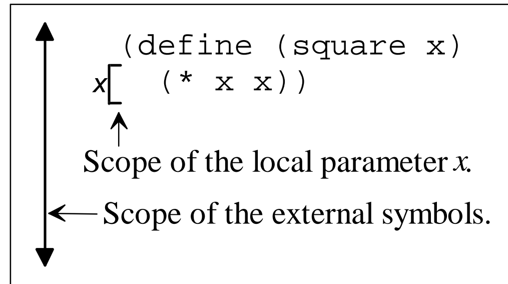
```
1. (#<(x) (if (= x 0) 5 #<*>)> 1)
2. (#<(x) (if (= x 0) 5 #<*>)> 1)
3. (if (= 1 0) 5 #<*>)
4. (if (#<=> 1 0) 5 #<*>)
5. (if #F 5 #<*>)
6. #<*>
```

The procedure was returned! This shows how procedures can be used as storage devices!

By Joshua Cantrell
jjc@cory.berkeley.edu

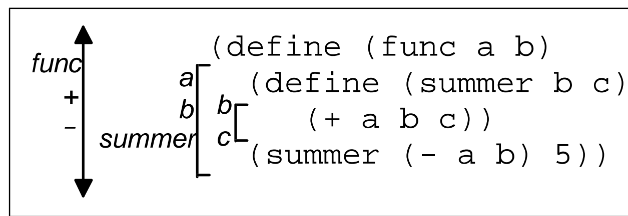
Lexical Scope[7]

Lexical scope describes the effective range of symbols bound inside and outside of procedures. It's named *lexical* because you can tell the scope of the bindings just by looking at the lexical organization of the code. As an example, the infamous *square* procedure:

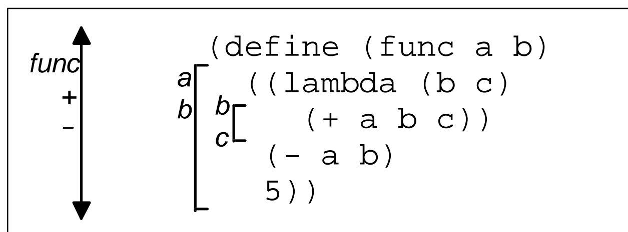


To show scope, a bracket can be drawn at the start and end of the region under influence and the symbols redefined within that scope can be drawn outside of the bracket as shown above. Notice that the scope of the symbol *x* only applies within the body of the procedure.

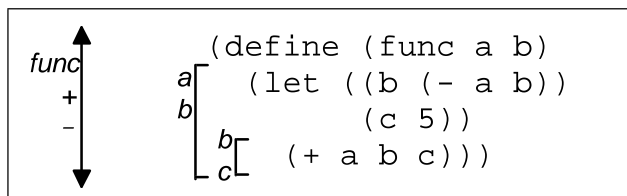
A more complex example would be helpful in understanding how scope works. Here's an example with procedures defined within procedures:



Notice how both the body of *func*[8] and the body of the internal procedure *summer*, share the same symbol *a*. They don't share the same symbol *b* since each one has its own definition of *b*. *C* is only defined within the procedure *summer*. A procedure that does the same thing can be written as follows:



Not binding a symbol to a procedure can be seen as having *no effect* on the scope of a procedure. This procedure can also be rewritten using a *let* special form.



---

[7] Lexical scope is also known as static scope.

[8] When I name a procedure by the symbol bound to it, I don't mean that the procedure *is* that symbol, I'm merely trying to save space by referring to it by that name. Always remember I'm talking about the procedure, not the symbol.