Implementing a simple substitution evaluator for a Scheme like λ-calculus language in Scheme

© Matt Pedersen (matt@cs.unlv.edu), University of Nevada, Las Vegas, 2007

Preliminaries

Before we start writing code we have to agree on what we are implementing. This document will guide you through how to implement a simple Scheme evaluator for a subset of scheme (roughly equivalent to the λ-calculus [Kle35]) and according to The Rules of Evaluation Version 2 from lecture [Ped07].

Let us first try to understand the background, that is, let us introduce the λ-calculus.

The λ-calculus

The lambda-calculus was first introduced in 1935 by Church and Kleene, and forms the basis for a number of functional programming languages such as ML, Lisp and Scheme.

The basic rules are pretty simple, and can be summarized as follows:

A lambda-expression is defined inductively as one of the following

1. `V`, a ***variable***, where `V` is any identifier (The precise set of identifiers is arbitrary, but must be finite).
2. `(λ V. E)`, an ***abstraction***, where V is any identifier and E is any lambda expression. An abstraction corresponds to an anonymous function.
3. `(E E′)`, an ***application***, where `E` and `E′` are any lambda expressions. An application corresponds to calling a function `E` with arguments `E′`.

We need to consider a few more issues, such as substitution, but we will return to those in a later section.

As you can see from the 3 rules above, the ability to declare a function abstraction and to apply it, as well as the ability to reference variables is all that is needed. That does not sound like much, but this is actually a Turing-complete language!

The Scheme Rules of Evaluation Version 2

Let us briefly review the rules of evaluation as they are stated in version 2 from lecture:

- **[Initialization Rule]:** The environment initially contains only the built-in primitives.
- **[Number Rule]:** A numerals value is a number interpreted in base 10.
- **[Name Rule]:** A name is evaluated by substituting the value bound to it in the environment.
    - It does not matter if it is user defined or a name of a primitive procedure.
    - If the name does not exist in the environment when the form is evaluated, an error is produced.
- **[Lambda Rule]:** The value of a lambda-form is a procedure with parameters and a body.
- **[Application Rule]:** An application is evaluated by evaluating each of its elements (Using the Rules) and then use
    - The Primitive rule if the operator is a primitive.
    - The Procedure rule if the operator is a procedure.
- **[Primitive Rule]:** Invoke the built-in primitive with the given arguments
- **[Procedure Rule]:** A procedure application is evaluated in 2 steps:
    - In the body of the procedure, replace each of the formal parameters by its corresponding actual arguments.
    - Replace the entire procedure by the body.
- **[Definition Rule]:** The 2nd argument is evaluated. The 1st is not evaluated and must be a name. The name/value pair is added to the environment.

As an aside, it is worth noting that the Initialization and the Definition rules are really not necessary; if we do not have definitions, then we do not need an environment, and definitions are not technically necessary; at least not in order to preserve Turing completeness. The Number and Primitive rules can be discarded on the same account.

One might say that a calculus that does not have numbers is not much fun, but in pure lambda-calculus, number can be represented as higher-order functions, something which a little hairy and we shell not bother with that at this moment.

You might say that if we remove the environment, then how can the Name rule exist? It is clear that the name rule corresponds to the first rule of the lambda-calculus definitions. We shall see how substitution can also solve this problem later. Furthermore, the Lambda Rule corresponds to the second point in the definition of the lambda-calculus, and the Application (and the Procedure) Rules correspond to the third part of the definition.

With that in mind we can now start coding the evaluator.

The Scheme Evaluator

To make things a little clearer in steps to come I have decided to place all primitives in a 'primitive environment' and only use the global environment for user-defined bindings.

The task ahead can be defined as follows:

> "Write a Scheme function called `evaluate`, which takes in a form and evaluates it using the global environment according to the Rules of Evaluation Version 2."

Thus the start of the evaluate function must look like this:

```
(define evaluate
  (lambda (form )
    ...
```

We have not discussed what a form is in our evaluator, but since all Scheme forms are either a variable or a value, or something with ( ) around, then a natural choice is to define a form in our evaluator as

- A symbol representing a variable or a value
- A list representing either a lambda-form, an application or a define form.

Note how this definition is very close to the definition of a lambda expression given earlier . We will return to the list part of the above definition a little later.

The environment

The first step is to decide how to represent the environment. Recall that the environment is a collection of bindings of values to names, so it can be viewed as a list of pairs whose first element is the name and second element is the value bound to that name.

In Scheme the primitive `cons` (which takes two arguments) constructs a pair:

```
➢ (cons 1 2)
(1 . 2)
```

The first element of a pair can be accessed using the Scheme primitive `car`, and the second by the primitive `cdr`. A note of caution: If `x` is a pair then `(cdr x)` is an element. If `x` is a list then `(cdr x)` is the list with the first element removed:

```
➢ (car (cons 1 2))
1
➢ (cdr (cons 1 2))
2
```

```
➢ (car (list 1 2))
1
➢ (cdr (list 1 2))
(2)
```

We can thus represent the global environment as a **list of pairs** (We start with the empty environment):

```
(define env '())
```

And we can define the Initialization Rule as follows:

```
(define init-rule
  (lambda ()
    (set! env '())))
```

That is, we use the `set!` primitive to redefine the global variable `env`.

The primitive environment is defined in the same way (though here we do not need any initialization rule as it always stays the same):

```
(define prim (list (cons '+ +)
                   (cons '- -)
                   (cons '/ /)
                   (cons '* *)
                   (cons '= =)
                   (cons '> >)
                   (cons '< <)))
```

In the list of primitives above I have included just a few of the existing ones, but note the content of the various pairs: the fist element (the `car`) is a symbol representing the operator (it is quoted thus a symbol), but the second element of the pair is not quoted, so it gets evaluated by the `cons` function before the pair is created. Let us try typing one of the pairs into the Scheme evaluator:

```
➢ (cons '+ +)
(+ . #<primitive:+>)
```

And now consider this:

```
➢ ((cdr (cons '+ +)) 4 5)
9
```

The second element is actually a primitive procedure that can applied by providing arguments and placing application brackets around it. At this point it is worth considering the following problem though: We know that we can apply the actual primitive to one or

more arguments by placing application brackets around the primitive and the argument(s); but what do we do if we have a list of arguments and a primitive, say we wish to apply the primitive function + to 4 and 5 but 4 and 5 appear in a list as `(4 5)`? Let us try:

```
➢ (+ (4 5))
procedure application: expected procedure,
                       given: 4; arguments were: 5
➢ (+ '(4 5))
+: expects argument of type <number>; given (4 5)
```

That the first one fails is not a surprise! `(4 5)` is an application of 4 to 5! That does not make any sense! But what about the second one? + takes 1 or more arguments (actually, `(+)` is actually a legal form in Scheme, it gives the element that is neutral to the operator, i.e., the value 0 for + and 1 for * and so on). The arguments must be something that evaluates to numbers, but `'(4 5)` is not a number, it is a list!

The solution is the Scheme function apply. Apply takes a function and a list and applies the function to the elements in the list in the following way:

$(\text{apply f '}(e_1\ e_2\ e_3\ \ldots\ e_n)) := f(\ldots(f(f(e_1\ e_2)\ e_3)\ldots)\ e_n)$

Or in scheme notation:

$(f\ (\ldots\ f((f\ e_1\ e_2)\ e_3)\ \ldots)\ e_n)$

So we get:

```
➢ (apply + '( 1 2 3 4))
10
```

Now that we have this little trick in place let us move on to the next rules.

The Number Rule

The Number rule is easy! The Scheme evaluator will do all the work for us; In reality the Number rule can be implemented simply as the identity function:

```
(define number-rule
  (lambda (number)
    number))
```

We simply return the argument! The scheme evaluator (because it is call by value!) will have already turned the textual representation of any number into an actual value by the time the body of the Number rule is executed.

So far it has not been too complicated, so let us turn to the Name rule.

The Name Rule

The Name rule will be a little more complicated because I decided to have two environments: one with user defined bindings and one with primitives. It will not complicate matter too much though. Recall that both the primitive environment and the global user environment is a list of pairs. We could write a function that takes in an environment (a list of pairs) and a symbol (the name we are looking for) and recursively traverses the list of pairs, but scheme has a built in function called `assv`, which can be utilized in the following way:

```
➢ (assv '+ (list (cons '+ +) (cons '- -)))
(+ . #<primitive:+>)
➢ (assv 'a '((b 5) (c 7)))
#f
```

`assv` takes in a symbol `x` and a list of pairs, and if a pair `(x . y)` exist it returns `(x . y)`. If no such pair exists it returns `#f`.

If we search the global environment first and in case we do not find anything there we can search the primitive environment, then the code looks like this:

```
(define name-rule
  (lambda (name)
    (let ((pair (assv name env)))
      (if pair
        (cdr pair)
        (let ((pair (assv name prim)))
          (if pair
            (cdr pair)
            (error (format "reference to undefined
                           identifier: ~a" name)))))))))
```

First we search the user environment; if nothing is found there we search the primitive environment; if still nothing is found we produce an error and terminate. Note, since the global environment is really global, we do not need to take a parameter representing the environment; if we do, we might run into problems later when writing the function for `define`. This is not the most elegant way of doing it, but it will work for now; later we will look at the **right** way of doing it (it works just fine when we are not implementing a language that supports commands like `set!` and captured frames!).

The Primitive Rule

Before we continue with the Primitive Rule, it is vital that we understand what data we are working with! Let us consider the Scheme form (+ 4 5). How do we represent that textually in our evaluator? The easiest ways is a list with 3 elements: the symbol + , the number 4 and the number 5 (In Scheme any number quoted becomes just the number) This can be illustrated by the use of the `equal?` predicate:

```
➢ (equal? '(+ 4 5) (list '+ '4 '5))
#t
```

So in our evaluator + will be looked up in the primitive environment and turned into the Scheme primitive `#<primitive:+>` and the list of arguments (which could be more complex forms than just integers) will be evaluated, and we end up with a list of values. Now we need to apply the primitive to the list of values. This is where the use of the apply function is handy, if the function to evaluate the Primitive rule takes in a primitive and a list of evaluated forms (i.e., a list of values) then we can apply that primitive to the list of arguments by using the `apply` function:

```
(define primitive-rule
  (lambda (primitive arguments)
    (apply primitive arguments)))
```

For example. If `primitive` is `#<primitive:+>` and `arguments` is `(4 5)`, then we have:

```
(apply #<primitive:+> (4 5)) = (#<primitive:+> 4 5)
```

which is exactly what we wanted! (Note, you cannot type the first part of the line above into the Scheme evaluator, I just wrote it to illustrate a point!)

The Application Rule

The Application rule is simple; it consists of two major parts:

1. Evaluate all the arguments of the form.
2. Use either the Primitive Rule or the Procedure Rule to evaluate the form.

We have already described the Primitive Rule; recall it takes in a Scheme primitive (obtained from a look up in the primitive environment) and a Scheme list of (evaluated) values.

How do we evaluate all the arguments of the form? Since we are writing (or at least have started) the evaluate function we can call that function of every element of the list of the application and then decide if we should pass it to the function that handles the Primitive

Rule (`primitive-rule`) or the one that handles the Procedure Rule (which we have not written yet!).

Let us consider the Scheme list `(+ (+ 4 5) 6)`. Before the Application Rule decides to pass the job on to the Primitive Rule, it must evaluate the arguments of the application, In this case there are 3 arguments: `+`, `(+ 4 5)`, and `6`. If we pass each of these three parameters (which technically are forms themselves) to the evaluate function then we should get 3 values back, one for each, and if we then collect them back into a list of length 3 (just like the original list for the application) we should get `(#<primitive:+> 9 6)`, which we can then pass on to the `primitive-rule` function for further evaluation. In the implementation we pass the primitive `(car evaluated-args)` and the arguments `(cdr evaluated-args)` in two separate parameters, but the idea is the same. In other words, we wish to evaluate the call `(evaluate ...)` for each of the members of the application form.

Again, we could write a recursive function that takes in the application form, calls `evaluate` on each element by taking out the first element (using the `car` function) and `cons`'ing that together with the result of a recursive call on the tail of the list (using the `cdr` function). Again Scheme already has a primitive (a very powerful one that is) called `map`, which can do all the dirty work for us.

`map` is a function that takes a function and a list of elements and applies the function to all the elements of the list and returns a new list with return values of this operation:

```
➢ (map add1 '(1 2 3))
(2 3 4)
```

Or if you wish to use one of your own functions:

```
➢ (map (lambda (x) (* x 3)) '(1 2 3))
(3 6 9)
```

We can formally define map like this:

$$(\text{map } f\ '(e_1\ e_2\ e_3\ \ldots\ e_n)) := ((f\ e_1)\ (f\ e_2)\ \ldots\ (f\ e_n))$$

In reality, `map` can do much more, but in the spirit of Terry Pratchett's Discworld, we call the above explanation 'lies to children'; it is enough information for now; it work as it is, so don't ask! Enough to say that the following Scheme form is valid as well:

```
➢ (map (lambda (x y) (+ x y)) '(1 2 3) (4 5 6))
(5 7 9)
```

It could be simplified to `(map + '(1 2 3) '(4 5 6))`.

This means that we can use the map function to evaluate all elements in a list and return the resulting values in a new list as follows (we assume that `lst` is a list representing the entire application):

```
(map evaluate lst)
```

For completeness, an implementation of map could look like this (one that takes in one function and one list (so not as fancy as the built-in one)):

```
(define map
  (lambda (f lst)
    (if (null? lst)
      '()
      (cons (f (car lst)) (map f (cdr lst))))))
```

Now we know how to evaluate the arguments, all we need to do now is determine if the first element in the application is a primitive or user defined procedure. This is easily done by looking for the textual representation (i.e., before evaluating the elements of the application) in the primitive environment. We simply use `assv` to search the primitive environment. If the first element is a lambda form or a named procedure (from the user environment, the lookup in the primitive environment will fail and we know that we should use the Procedure rule,; if it succeeds we can pass the application to the Primitive Rule for further evaluation. Thus the Application rule can be implemented as follows:

```
(define application-rule
  (lambda (lst)
    (let ((primitive? (is-primitive? (car lst)))
          (evaluated-args (map evaluate lst)))
      (if primitive?
          (primitive-rule (car evaluated-args)
                          (cdr evaluated-args))
          (procedure-rule (car evaluated-args)
                          (cdr evaluated-args))))))
```

To improve readability we can implement the `is-primitive?` predicate in the following way:

```
(define is-primitive?
  (lambda (e)
    (assv e prim)))
```

Recall that `assv` will return `#f` if nothing is found (i.e., there is no pair in the list `prim` that has `e` as the first element).

The next step should be the Procedure Rule, but since this is by far the most complicated rule, we shall put that off for just a little while. Let us instead turn to the Define Rule, which is a lot simpler.

The Define Rule

The Define rule says to evaluate the 2nd form (well, really, it is the 3rd because the word `define` is really the 1st) and bind it to the name given in the 1st form (which then is really the 2nd!) of the define form. The evaluation of the 2nd (which is really the 3rd) form is as simple as calling the evaluate function with this form:

```
(define define-rule
  (lambda (name form)
    (let ((value (evaluate form))
          (pair (assv name env)))
      (if pair
          (set-cdr! pair value)
          (set! env (cons (cons name value) env))))))
```

At first sight, the rule looks a little more complex then described above! We need to determine if the name is already bound in the environment. If it is then we simply change the second element of the pair returned from `assv` (the lookup in the environment). If there is no existing binding we set the environment to what it was before with the new name/value pair added to the front of the list representing the global environment.

If you are not familiar with `set!`, `set-car!` and `set-cdr!`, then see the discussion at the end of this write up.

We now have everything we need except the main procedure (`evaluate`) and an implementation of the Procedure Rule, so let us look at the Procedure Rule.

Procedure Rule

Before we implement the `procedure-rule` function we need to understand the rule in detail. The way it is written in The Rules Version 2 is not entirely correct! It could work as stated under one condition: **no** two procedures (bound in the global environment) or anonymous lambda forms share **any** parameter names!

Let us consider this small example:

```
((lambda (x) (+ ((lambda (x) (* x 2)) (+ x 4)) 5)) 1)
```

What is the value of such a form? Let us blindly use the procedure rule as stated in Version 2 of the Rules of Evaluation. The first part says to replace each of the formal parameters with its corresponding actual ones. This can be done by a substitution of the name of the formal parameter in the list representing the body of the function being applied with the value of the actual parameter. In the example above if we perform a substitution of `x` by the value 1, and look at the body, we get:

```
(+ ((lambda (x) (* 1 2)) (+ 1 4)) 5)
```

which if typed into the Scheme evaluator gives us the value 7. This is clearly wrong. If we type the original form into the evaluator we get the value 15, which is the correct value. The problem is of course the 'mindless' substitution of `x` by 1. This substitution should only happen for a certain number of the `x`'s found in the body, namely the ones that are 'free' (we will get to this definition shortly). If you look at it and think about it you would probably arrive at the conclusion that only the underlined `x`'s in the following should be substituted by the value 1:

```
((lambda (x) (+ ((lambda (x) (* x 2)) (+ x 4)) 5)) 1)
```

So only the last `x` should be substituted because the inner `x` is bound by the formal parameter in the inner most anonymous lambda form.

Let us look at the formal definition of a free variable in a lambda expression.

Free and Bound Variables

Each variable in a lambda expression is either ***free*** or ***bound***. For example the `x` in `(x y)` is free, while the `x in (lambda (x) (x y))` is bound. A bound variable has a specific lambda with which it is associated, while a free variable does not. The free variables of a lambda expression are defined inductively as follows (we define this for the lambda calculus first):

1. In an expression of the form `V`, where `V` is a variable, this `V` is the single free occurrence.
2. In an expression of the form (λ `V. E`), the free occurrences are the free occurrences in `E` except for `V`. In this case the occurrence of `V` in `E` are said to be bound by the λ before `V`.
3. In an expression of the form `(E E′)`, the free occurrences are the free occurrences in `E` and in `E′`

In general all formal parameters of a lambda form 'binds' any occurrences of that parameter in the body, making it not free (i.e., bound).

We can now define formally the correct rules for substitution.

Substitution

Substitution, written `E[V := E']`, corresponds to the replacement of a variable `V` by expression `E'` every place it is free within `E`. The precise definition must be careful in order to avoid accidental variable substitution. For example, it is not correct for `(λ x.y)[y := x]` to result in `(λ x.x)`, because the substituted `x` was supposed to be free but ended up being bound. The correct substitution in this case is `(λ z.x)`.

The precise rules are defined inductively as follows:

1. `V[V := E] == E`
2. `W[V := E] == W`, if `W` and `V` are different.
3. `(E1 E2)[V := E] == (E1[v := E] E2[v := E])`.
4. `(λ W. E')[V := E] == (λ W. E'[V := E])`, if `V` and `W` are different and `W` is not free in `E`.
5. `(λ W. E')[V := E] == (λ W'. E'[W := W'])[V := E]`, if `V` and W are the same (i.e., not different) and if `W'` is not free in `E`.

We did not include a rule for arbitrarily substituting the formal parameters, and the rules above are only defined for lambda expressions with one parameter, but multi parameter expressions follow trivially.

Note that rule 5 solves the problem of accidental variable substitution. Rule 1 and 2 explain how to substitute simple named variables, rule 3 is for applications and rules 4 and 5 are for lambda forms. Also note that substitutions only happen for free variables!

We can now rewrite the Procedure Rule as follows:

Old version:

- [Procedure Rule]: A procedure application is evaluated in 2 steps:
    - In the body of the procedure, replace each of the formal parameters by its corresponding actual arguments.
    - Replace the entire procedure by the body

New version:

- [Procedure Rule]: A procedure application is evaluated in 2 steps:
    - In the body of the procedure, replace each of the **free** formal parameters by its corresponding actual arguments.
    - Replace the entire procedure by the body

Note the slight difference; the word **free** has been added to the new version. You might think that there is no difference because the formal parameters in the body are bound by the enclosing lambda form; yes, you are right, but we are not making the substitution in

the entire lambda form, only in the body, where the formal parameter list cannot be seen! A small but very important difference.

So if we are considering an application like this:

```
(E E')
```

where `E` is a lambda form and `E'` is a set of actual parameters, we can expand as follows:

$$((\text{lambda } (p_1\ p_2\ \ldots\ p_n)\ E'')\ f_1\ f_2\ \ldots\ f_n)$$

We now evaluate the forms $f_1, f_2, \ldots, f_n$, to produce the values of the actual parameters $v_1, v_2, \ldots, v_n$, and then perform the substitution

$$E''[p_1 := v_1,\ p_2 := v_2, \ldots,\ p_n := v_n]$$

and then evaluate this new form!

We could implement the above rules in great detail and compute sets of free and bound variables, but there is a simpler (!) way, which we shall utilize.

Let us first write the `procedure-rule`, and make use of a few helper functions:

```
(define procedure-rule
  (lambda (lambda-form args)
    (evaluate (subst lambda-form args))))
```

Looks pretty simple! The one line in the body simply calls the `evaluate` function recursively with the result of the call to the helper function `subst` called with the lambda form and the list of evaluated arguments. `subst` returns the body of the lambda form after performing the substitution.

The `subst` helper will do the work of substituting all free occurrences in the body of the lambda form according to a list of pairs where each formal parameter is bound to the corresponding actual parameter value.

Since all procedure calls are call-by-value we never substitute any variable names by other variable names, only by values, which makes life a lot easier when it comes to getting the substitution correct.

Before we look at the implementation of `subst`, let us implement another helper function that is often handy to have, namely the `filter` function. The filter function takes a predicate function (a function that takes in one argument and returns either `#t` or `#f`) and a list of elements that are compatible with the function passed to filter.

Example:

```
➢ (filter odd? '(1 2 3 4 5 6 7 8 9))
(1 3 5 7 9)
```

Where the `odd?` function could be defined as follows:

```
(define odd?
  (lambda (x)
    (= (remainder x 2) 1)))
```

Where `(remainder x y)` returns the remainder of the integer division x/y.

The code for the `filter` function looks like this:

```
(define filter
  (lambda (predicate? lst)
    (if (null? lst)
        '()
        (if (predicate? (car lst))
            (cons (car lst)
                  (filter predicate? (cdr lst)))
            (filter predicate? (cdr lst))))))
```

which is a typical structure for a list processing function.

We can now write the subst function, which will call another helper with the body of the lambda form as well as a list of pairs that link the formal parameter of the procedure call to the actual ones. We can create this list by using the map function again. Consider this example:

```
➢ (map (lambda (x y) (cons x y)) '(x y z) '(1 2 3))
((x . 1) (y . 2) (z . 3))
```

If `actual-parameters` is a list of actual parameters, and `formal-parameters` a list of formals (which can be obtained as the second element of the lambda form) we can create a list of pairs and call the helper like this:

```
(define subst
  (lambda (lambda-form actual-parameters)
    (let* ((formal-parameters (cadr lambda-form))
           ((assoc-args (map (lambda (x y) (cons x y))
                             formal-parameters
                             actual-parameters)))
      (substitute (caddr lambda-form) assoc-args))))
```

Note the use of `let*` rather than `let`, this is necessary because the value of formal-parameters is used in the second binding in the `let*` form.

So now we pass the problem on to the `substitute` function, but at least we are getting there.

The `substitute` function will implement the lambda calculus substitution rules (more or less) , so we need to consider the different cases described above. Let us consider the `substitute` functions first few lines (recall that it will take in a form and a list of pairs representing the variable bindings of the actual parameters to the formal ones):

```
(define substitute
  (lambda (form bindings)
```

We now consider the 3 different types of forms that we can have:

1. A value or a variable `V`.
2. An applications `(E E')`.
3. A lambda form `(λ E. E')`.

We are going to use a little trick to deal with variables and values; we shall return to that shortly.

The easiest of the three cases is probably the application. Recall that an application is represented as a Scheme list, and according to the substitution rules above we simply perform a substitution in all of the elements in this list. Such a list can contain both variables, values or more complex forms (which are represented as lists, and can be dealt with by a recursive call to substitute). If an element is not a list, we check to see if it has a binding in the binding list, and if it does we substitute the variable name by the corresponding value. The easiest way to do this for each element of a list if of course to use the map function:

```
(map (lambda (element)
       (if (list? element)
         (substitute element bindings)
         (let ((pair (assv element bindings)))
                (if pair
                  (cdr pair)
                   element)))) lst)
```

The above action is what we wish to perform if `form` is a list and not a lambda form. What if the form is not a list, that is, a variable or a value. The easiest way to deal with this case is to place the variable or value in a list and call recursively and then extract the value out of the returned list using the `car` function; then the last part of the code above will take care of it, so we get:

```
(car (substitute (list form) bindings))
```

All we have left to do is to deal with the substitution for lambda forms, and then tie it all together. Remember, the first time the substitute function is called, it will be with a binding list and a body of a procedure, so if this body is a lambda form we only want to substitute the free variables in the this lambda form, or in other words, if this lambda form has any formal parameters in common with the enclosing lambda form, then we want to assure that these do not get substituted incorrectly. We assure this by making a recursive call on the body of the lambda form (which formed the body that we are considering in the first place) but with any binding to a formal that also occurs in the inner lambda form removed. Once we get the result back we can reconstruct the lambda form and return it:

```
(list 'lambda
      (cadr form)
      (substitute (caddr form)
                  (remove-binding bindings
                                  (cadr form))))
```

The entire `substitute` procedure thus looks like this:

```
(define substitute
  (lambda (form bindings)
    (if (list? form)
      (if (eq? (car form) 'lambda)
        (list 'lambda
              (cadr form)
              (substitute (caddr form)
                          (remove-binding bindings
                                          (cadr form))))
        (map (lambda (element)
               (if (list? element)
                 (substitute element bindings)
                 (let ((pair (assv element bindings)))
                   (if pair
                     (cdr pair)
                     element)))) lst))
       (car (substitute (list form) bindings)))))
```

Now we just need to consider the remove-binding procedure. All it needs to do is return the original binding list with any binding whose name is equal to any of the names in the formal parameter list removed:

```
(define remove-binding
  (lambda (assoc-lst name-lst)
```

```
    (for-each
      (lambda (name)
        (set! assoc-lst
              (filter (lambda (pair)
                        (not (eq? (car pair) name)))
                      assoc-lst)))
      name-lst)
    assoc-lst))
```

We use the `for-each` function, which behaves just like map, except it does not return anything. For each element in the list of names that we wish to remove from the binding list we perform the `(set! ...)` line, which updates the binding list with the result of the call to filter. In other words, each call to `filter` will either return the original binding list, or the original binding list with one pair removed, namely the pair `(name . ???)` if it exists in the list. We do this filtering for each name, and update the binding list each time using `set!`, once we are done we return the new binding list.

With everything in place we can now write the `evaluate` function.

The Main Evaluate Function

A Scheme form is either of the form `V`, where `V` is a name or a number, or of the form `(...)` which is a list. We consider these two different formats as the first part of the evaluate function.

```
(define evaluate
  (lambda (form)
    (if (list? form)
      ;; case statement here
      ;; if statement here
     )))
```

Let us deal with the if statement first:

Either the form is a number (in which case we call the `number-rule` function) or it is a name (in which case we call the `name-rule` function)

```
(if (number? form)
  (number-rule form)
  (name-rule form))
```

The case statement looks like this:

```
(case (car form)
  ((define) (define-rule (cadr form) (caddr form))))
  ((lambda) form)
```

```
((if)      (if-rule (cadr form)
                    (caddr form)
                    (cadddr form)))
(else      (application-rule form)))
```

Note that in our evaluator we really do not do anything to evaluate a lambda form (unless it is being applied!), that is why we do not have a lambda-rule like in The Rules of Evaluation. Note, I put an If Rule in, we will describe this rule shortly.

Lastly, we can write a loop to constantly read and evaluate input from the user:

```
(define loop
  (lambda ()
    (display "> ")
    (let ((command (read)))
       (if (eq? command 'stop)
         #t
         (begin
           (let ((result (evaluate command)))
             (if (not (void? result))
               (display (format "Result: ~a" result)))
             (loop)))))))
```

This concludes this little exercise in writing an evaluator. Keep in mind that there is no syntax checking, which of course means that the evaluator will crash if you do not feed it syntactically correct programs. You can restart it with the command `(loop)`.

Additional Rules

If Rule

A programming language is not much fun if we cannot make choices. So for completeness we add an if statement (just like the if statement in Scheme).

The If Rule is simple:

> Syntax: `(if form1 form2 form3)`
>
> [If Rule] Evaluate `form1`, if it does not evaluate to `#f` then evaluate `form2` else evaluate `form3`.

```
(define if-rule
  (lambda (test-form true-form false-form)
    (if (evaluate test-form)
      (evaluate true-form)
      (evaluate false-form))))
```

A small curiosity about the If Rule is that **only one** of either the `true-form` or the `false-form` is evaluate. This can be seen in Scheme by the following:

```
➢ (define a 1)
➢ (define b 1)
➢ (if #f (set! a 2) (set! b 2))
➢ a
1
➢ b
2
```

This means that an if form is not evaluated as a regular application. We call an if form a ***special form*** because not all elements are evaluated. The define form is also special as the name part is not evaluated (i.e., not attempted looked up in the environment before it is bound!)

Global vs local environments

Consider the following Scheme forms:

```
➢ (define x 1)
➢ (define f (lambda (x)
              (lambda (y)
                (set! x (+ x y))
                x)))
➢ (define g (f 0))
➢ x
1
➢ (g 10)
10
➢ (g 0)
10
➢ x
1
```

We first define a global variable `x` (bound in the global user environment) and assign it the value 1. Next we define a function `f`, which takes in a parameter `x` and returns a procedure that accepts one parameter `y`, adds `y` to `x` and returns `x`.
We then define `g` to be the function returns from a call to `f` with the value `0` for the parameter `x`.
When we make the call `(g 10)` the return value is `10` and not `11`! Why is that? The `x` used by `g` is not the `x` in the global environment, but the value of the parameter passed to `f` when `g` was bound. Thus this parameter `x` must live somewhere else than the global environment. In our version of the Scheme interpreter (the one we just wrote) we cannot emulate the above! We do not have `set!` in our implementation (`set!` is not a part of the λ-calculus). As a matter of fact, if we wish to support functions like `set!` (which alters an existing variable binding, then we cannot implement the evaluator with call-by-value substitution). We would need to consider the frames that get generated using the droid model presented in lecture. A frame is really an activation record, which holds all the bindings for the formal parameters, and sometimes these frames are not removed when the function call with which they were associated terminate. If a function returns another function that accesses some of the formal parameters like in the above example, the frame is 'captured' and kept around. For our evaluator to support such craziness we could need to associate with each binding in the global environment of a function an environment, namely the environment in which it was 'born'.

Most function would be born in the global environment if they are straight forward `(define <name> (lambda (…) <body))` function. However, if a function is bound and placed in the global environment (like g is above), but the function is 'born' in a different frame (in the case above the function bound to `g` is born in the frame associated with `f`), then we must associate this frame with the function. This gives us

what is commonly referred to as a ***static-link***. A static link is used for scope-resolution, that is, it tells us where to look for variables when we need them; we first look in the frame in which the function was born, then in its parent's frame etc. This mimics the well known scoping mechanism known as static scoping found in many languages like C and Java. A substitution based language like the one we just implemented does not have the notion of scoping apart from the global environment and the parameters in a lambda form.

This can get pretty hairy, so we will not consider the finer details of this problem, but if we wish to extend our evaluator to support `set!` like functions, we need to do the following:

1. Associate with each binding of a function the frame it was born in.
2. When applying a function to arguments, and when encountering a name, look for the name in the associated environment, if nothing is found, look in the environment of the closest enclosing lambda form. Keep doing this until the global user environment is reached, and if the name is not defined in this chain of environments, then the name is not bound.
3. Drop the substitution method for the Procedure Rule. Instead, the Procedure Rule creates a new environment, fills it with bindings and then evaluates the body.
4. Change the Name Rule to work in accordance with 2.

For completeness it should be noted that `set-car!` and `set-cdr!` works just like `set!` just operates on pairs and sets the `car` or the `cdr`. Furthermore, it should be clear that since a function like `set!` **alters** state, it breaks the substitution model as there are no variables kept when substituting the free variables in the body (except for the free variables that are not bound by the binding list of formals to actuals)

One last small curiosity arises when we do add an extra parameter representing the environment to all our evaluation functions. In particular consider the problem we get if we change the `evaluate` function to take in a form as well as an environment: In the Application Rule we map the evaluate function onto the list of forms like this `(map evaluate lst)`, but now we need to add an extra parameter to the `evaluate` function, namely the environment (let us call it `env`).

Let us experiment with `map`:

```
➢ (map evaluate lst)
map: arity mismatch for procedure evaluate: expects 2
     arguments, given 1
```

or

```
➢ (map evaluate lst env)
map: all lists must have same size; arguments were:
     #<procedure:evaluate> (4 5 6) ()
```

The last error message is a little confusing; it just happens that `env` is in fact a list, but even if `env` had the right length, it would all be wrong because subsequent uses of `env` would assume it to be a list of pairs, not just a pair! This can take some time for you to understand, but it is well worth spending a little time on; as I said, `map` is a very powerful and useful function to know and understand.

The correct way of doing it is as follows:

```
(map (lambda (form) (evaluate form env)) lst)
```

By wrapping an anonymous lambda form around the call to `evaluate`, we get a function that takes one parameter (which `map` will supply from the list `lst`), and calls `evaluate` with 2 parameters: the element supplied by `map`, and the environment.

Tracing

In the discussion of the implementation I did not mention anything about tracing the evaluator. It is an interesting ability to have, but it would have cluttered the description of the implementation, so I left it out. However, in my implementation I have added tracing.

Tracing can be turned on and off with the commands `(trace on)` and `(trace off)`. If you turn trace on you will get information about what the evaluator is currently evaluating.

Here is an example showing a trace of the factorial function:

```
> (define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
evaluating '(define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))' [Define Rule]
evaluating '(lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))' => #<procedure>

> (fact 3)
Result: 6

> (trace on)

> (fact 3)
evaluating '(fact 3)' [Application Rule]
| evaluating 'fact' => (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))) [Name Rule]
| evaluating '3' => 3 [Number Rule]
| [Procedure Rule]
| Substituting in '(if (= n 0) 1 (* n (fact (- n 1))))' with bindings: ((n . 3))
| | Substituting in '(= n 0)' with bindings: ((n . 3))
| | => '(= 3 0)'
| | Substituting in '(* n (fact (- n 1)))' with bindings: ((n . 3))
| | | Substituting in '(fact (- n 1))' with bindings: ((n . 3))
| | | | Substituting in '(- n 1)' with bindings: ((n . 3))
| | | | => '(- 3 1)'
| | | => '(fact (- 3 1))'
| | => '(* 3 (fact (- 3 1)))'
| => '(if (= 3 0) 1 (* 3 (fact (- 3 1))))'
| evaluating '(if (= 3 0) 1 (* 3 (fact (- 3 1))))' [If Rule]
| evaluating '(= 3 0)' [Application Rule]
| | evaluating '=' => #<primitive:=> [Name Rule]
| | evaluating '3' => 3 [Number Rule]
| | evaluating '0' => 0 [Number Rule]
| => #f [Primitive Rule]
| evaluating '(* 3 (fact (- 3 1)))' [Application Rule]
| | evaluating '*' => #<primitive:*> [Name Rule]
| | evaluating '3' => 3 [Number Rule]
```

```
| | evaluating '(fact (- 3 1))' [Application Rule]
| | | evaluating 'fact' => (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))) [Name Rule]
| | | evaluating '(- 3 1)' [Application Rule]
| | | | evaluating '-' => #<primitive:-> [Name Rule]
| | | | evaluating '3' => 3 [Number Rule]
| | | | evaluating '1' => 1 [Number Rule]
| | | => 2 [Primitive Rule]
| | [Procedure Rule]
| | Substituting in '(if (= n 0) 1 (* n (fact (- n 1))))' with bindings: ((n . 2))
| | | Substituting in '(= n 0)' with bindings: ((n . 2))
| | | => '(= 2 0)'
| | | Substituting in '(* n (fact (- n 1)))' with bindings: ((n . 2))
| | | | Substituting in '(fact (- n 1))' with bindings: ((n . 2))
| | | | | Substituting in '(- n 1)' with bindings: ((n . 2))
| | | | | => '(- 2 1)'
| | | | => '(fact (- 2 1))'
| | | => '(* 2 (fact (- 2 1)))'
| | => '(if (= 2 0) 1 (* 2 (fact (- 2 1))))'
| | evaluating '(if (= 2 0) 1 (* 2 (fact (- 2 1))))' [If Rule]
| | evaluating '(= 2 0)' [Application Rule]
| | | evaluating '=' => #<primitive:=> [Name Rule]
| | | evaluating '2' => 2 [Number Rule]
| | | evaluating '0' => 0 [Number Rule]
| | => #f [Primitive Rule]
| | evaluating '(* 2 (fact (- 2 1)))' [Application Rule]
| | | evaluating '*' => #<primitive:*> [Name Rule]
| | | evaluating '2' => 2 [Number Rule]
| | | evaluating '(fact (- 2 1))' [Application Rule]
| | | | evaluating 'fact' => (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))) [Name Rule]
| | | | evaluating '(- 2 1)' [Application Rule]
| | | | | evaluating '-' => #<primitive:-> [Name Rule]
| | | | | evaluating '2' => 2 [Number Rule]
| | | | | evaluating '1' => 1 [Number Rule]
| | | | => 1 [Primitive Rule]
| | | | [Procedure Rule]
| | | | Substituting in '(if (= n 0) 1 (* n (fact (- n 1))))' with bindings: ((n . 1))
| | | | | Substituting in '(= n 0)' with bindings: ((n . 1))
| | | | | => '(= 1 0)'
| | | | | Substituting in '(* n (fact (- n 1)))' with bindings: ((n . 1))
| | | | | | Substituting in '(fact (- n 1))' with bindings: ((n . 1))
| | | | | | | Substituting in '(- n 1)' with bindings: ((n . 1))
| | | | | | | => '(- 1 1)'
| | | | | | => '(fact (- 1 1))'
| | | | | => '(* 1 (fact (- 1 1)))'
| | | | => '(if (= 1 0) 1 (* 1 (fact (- 1 1))))'
| | | | evaluating '(if (= 1 0) 1 (* 1 (fact (- 1 1))))' [If Rule]
| | | | evaluating '(= 1 0)' [Application Rule]
| | | | | evaluating '=' => #<primitive:=> [Name Rule]
| | | | | evaluating '1' => 1 [Number Rule]
| | | | | evaluating '0' => 0 [Number Rule]
| | | | => #f [Primitive Rule]
| | | | evaluating '(* 1 (fact (- 1 1)))' [Application Rule]
| | | | | evaluating '*' => #<primitive:*> [Name Rule]
| | | | | evaluating '1' => 1 [Number Rule]
| | | | | evaluating '(fact (- 1 1))' [Application Rule]
| | | | | | evaluating 'fact' => (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))) [Name Rule]
| | | | | | evaluating '(- 1 1)' [Application Rule]
| | | | | | | evaluating '-' => #<primitive:-> [Name Rule]
| | | | | | | evaluating '1' => 1 [Number Rule]
| | | | | | | evaluating '1' => 1 [Number Rule]
| | | | | | => 0 [Primitive Rule]
| | | | | [Procedure Rule]
| | | | | Substituting in '(if (= n 0) 1 (* n (fact (- n 1))))' with bindings: ((n . 0))
| | | | | | Substituting in '(= n 0)' with bindings: ((n . 0))
| | | | | | => '(= 0 0)'
| | | | | | Substituting in '(* n (fact (- n 1)))' with bindings: ((n . 0))
| | | | | | | Substituting in '(fact (- n 1))' with bindings: ((n . 0))
| | | | | | | | Substituting in '(- n 1)' with bindings: ((n . 0))
| | | | | | | | => '(- 0 1)'
| | | | | | | => '(fact (- 0 1))'
| | | | | | => '(* 0 (fact (- 0 1)))'
| | | | | => '(if (= 0 0) 1 (* 0 (fact (- 0 1))))'
| | | | | evaluating '(if (= 0 0) 1 (* 0 (fact (- 0 1))))' [If Rule]
| | | | | evaluating '(= 0 0)' [Application Rule]
| | | | | | evaluating '=' => #<primitive:=> [Name Rule]
| | | | | | evaluating '0' => 0 [Number Rule]
| | | | | | evaluating '0' => 0 [Number Rule]
| | | | | => #t [Primitive Rule]
| | | | | evaluating '1' => 1 [Number Rule]
| | | | => 1 [Primitive Rule]
| | | => 2 [Primitive Rule]
```

```
| => 6 [Primitive Rule]
Result: 6
```

Syntax Checking

As mentioned earlier, the evaluator that we implemented does not have any syntax checking. This means that the evaluator will crash if you type in a syntactically illegal form.

I have added a crude syntax checker to my version of the evaluator. The syntax checker catches the most frequently made errors. It is still fairly easy to break it.

My Evaluator

You can download and play with my version, and improve it if you like. It can be obtained from the course website (www.cs.unlv.edu/~matt) under teaching/CS789.

References

[Kle35] Kleene, Stephen, *A theory of positive integers in formal logic*, American Journal of Mathematics, 57 (1935), pp. 153–173 and 219–244

[Ped07] Pedersen, Matt, *Lecture Notes for CS789 Fall 2007,* 2007