Anyone in computer science will encounter functions and variables in their different forms as they work their way through the joyous, but sometimes difficult world of programming. From language to language "variables" are known by different names and take several forms. In Scheme though, we do not use variables. We will be using something known as an identifier due to several reasons, but we will focus on the fact that variables in other languages do not usually have a set, unchanging value. In Scheme we will use these identifiers because the code being represented and explained will not allow the value to be changed, and identifiers have set values. When working with functions, identifiers will inevitably be encountered; which leads us to a burning question, what needs to be done to evaluate that function and get a value? Here lies a vital method of interpreting code known as substitution.

In its simplest form substitution is nothing more than plugging in values where the identifier matches the identifier being called in a function. This may sound convoluted, but is very simple, as seen in the example below.

x = 4  
y = x + 1  
y = 4 + 1  
y = 5

In this example 'x' identifies the number 4. When you look at the equation below it you will find that you need a value for 'x'. Substitution is nothing more than plugging that value in and solving the equation . Remember, this is a real world model of a programming concept, it is not meant to be code. In Scheme, assuming you cannot change the value of 'x' this problem is very simple and the input can easily be processed and return the resulting number (5). In our interpreter this simple case has to be handled, along with many other varying levels of complexity. We will be using two types of interpreters, one that directly substitutes values and another that completes the task by capturing the identifiers inside of an environment and later substitutes the values where they belong. The directly substitutive interpreter takes the values at the time they are encountered and calls a function that replaces them as the input is interpreted. With this method there is no need to store the identifiers and values they identify for later use as they are used immediately. An example this can be seen below, minus the working code of course as that can be lengthy.

{with {x {- 11 8}} {+ x x}}

As seen, there is a "with" structure being used. In this structure there is an identifier, the 'x' which is a binding. This means that 'x' is bound to the value, or expression as is this case "- 11 8", which if evaluated is just the number 3. The term binding is used because the value of 'x', once set, cannot be changed and all values of 'x' will be (- 11 8). Along with the binding, "with" structures contain a body. The body of the with structure is an expression which can be evaluated (or another with as I will explain in more detail later). When the interpreter sees this with structure it will look into the body and find that there is a value 'x'. From here the binding and body will be passed as parameters into a substitution function which will match the identifier 'x' with the identifier 'x' in the body and replace that identifier with its actual value.

{with {x {- 11 8}} {+ (- 11 8) x}}

This result will be passed back into the interpreter and it will find another identifier, which in this case will do the same as it did before. Now we are left with the result below.

{with {x {- 11 8}} {+ (- 11 8) (- 11 8)}}

Now that the body no longer has any identifiers remaining, the body of the result above will be passed into a function to be interpreted. In our interpreter this function will use the following code to render the expression down to a single numeric value.

...  
 [(number? sexp) (make-num sexp)]

[(symbol? sexp) (make-id sexp)]

[(equal? (first sexp) '+)

(make-binop (first sexp) (parse (second sexp)) (parse (third sexp)))]

[(equal? (first sexp) '-)

(make-binop (first sexp) (parse (second sexp)) (parse (third sexp)))]  
 ...

As this code evaluates the below expression it will make structures that will be evaluated down to their most simplified form - a single number in this case. This process would look something like the example below shows.

{+ (- 11 8) (- 11 8)}  
{+ 3 3}  
6

In some cases there may be nested with statements, such as the case in this example.

{with { x 6} {with {y 11} {+ x y}}

When this happens the body of the first with structure contains a second with structure. In this case the interpreter will move on to the body of the second with and will find the identifier 'y' and will substitute that with the 'y' in the previous with. The 'x' will then be replaced with the 'x' in the current with. This is all done in the same manner as the previous example was, the ordering and function calls will just be handled slightly differently.

As mentioned before there were two types of interpreters, we just discussed a directly substitutive interpreter, but have yet to cover one that uses environment capturing. With this method, instead of immediately replacing the identifiers with their actual values and then solving the resulting expression we will be storing the binding identifiers in an environment. This is just a list of bindings being stored for later reference. Instead of walking through the expression and substituting as it is traversed the program will store all of the identifiers in the environment and just leave the body of the statement and that accompanying environment remaining.

{with {c 3} {+ c 3}}  
{with {c 3} {+ c 3}} Env: '({c 3})  
{+ c 3} Env: '({c 3})  
{+ 3 3}  
6

One can see that the binding above is stored in an environment and the with structure is parsed; the resulting expression is now ready for substituting the values which were stored in the environment. Once again working recursively, the body and environment will be passed back into the method and the identifier in the body will be searched for in the environment by recurring on the rest of the environment until a match is found. If this is the case the corresponding value will be substituted into the body and a result will be returned. If the identifier cannot be matched with one inside the environment then that signifies an invalid input and an error will be returned as it us unable to be processed further.

Moving on to functions, our interpreter must handle things a bit differently. To begin, we are no longer able to just add our bindings to the environment. If this was the case there would never be a useable match from within the environment. As seen in the example below if the binding (f 3) would be added to the environment, we would be unable to parse the function any further.

(f (fun (x) (+ x 5))(f 3)

This is because the interpreter will search the environment for a matching identifier, which in this case is 'x'. This identifier will not match anything currently in the environment as it contains (f 3). To fix this we will have to pass the function value '3' into the function and bind this value to the identifier 'x'. Once this process is complete the binding will be stored in the environment for later use.

In addition to handling functions differently with respect to the environments, if the current environment is always interpreted one will find that in some cases the interpreter will fail. If you have a binding, say (x 5) and your expression (+ x 3) followed by another binding of (x 8) and the current environment is always interpreted this will cause an incorrect result. Note that there are two of the same identifiers but they each have different values. If we interpreted this using the current environment the result would be 11, but, if this was interpreted using the environment that existed at the time the expression was initialized the result would be 8. This means our interpreter must always evaluate an expression using the environment that existed when it was initialized, and not the current environment.

(with (f (fun (x) (+ x 5))  
(with (x 1000)  
(f 3)))

If the above code was evaluated from the current environment, the value of 'x' would be evaluated assuming it was 1000. The final answer would come out to be 1005 as the current environment is (x 1000) and not (x 3) as it should be. By interpreting this with the environment it was initialized with, the number returned would be 8. In this case the environment has been properly handled and the old environment has been evaluated. This is done by capturing the old environment and recalling it at the time of evaluation, so, even if the environment has changed it will still look at the environment as it was when the function was initially processed, ensuring consistency. Up until this point our environment was just a list of bindings, but we now need a way to capture these "current environments". To do this we must add a second field to our list of environments. Now there will be a binding linked to the environment that was present at the time this new environment was stored. This will allow us to recall a previous environment state and always ensure we are interpreting our expressions with the correct environment.

Direct substitution and environment capturing interpreters make processing functions and expressions a simple process, but the order of evaluation and type of data can be confusing at best. Through direct substitution values are just plugged into where the identifiers match. With environments a binding is captured and stored for later use and consists of an id and a value. When an ID is matched the value is pulled from the binding and is interpreted. Functions are merely an extension of our environment capturing interpreter and allows parameters to be passed into the function and enclosed expressions. These parameters correspond to an ID that is defined in the function. When this ID is matched the value will, as before, be pulled from the binding and used to evaluate the function.