Programming that makes extensive use of assignment is known as *imperative* programming.

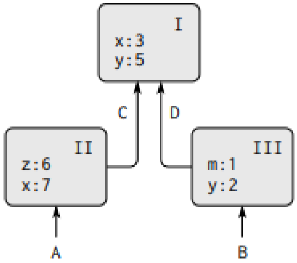
Programming that makes extensive use of assignment is known as *imperative* programming.

In the presence of assignment, a variable can no longer be considered to be merely a name for a value.
Rather, a variable must somehow designate a "place" in which values can be stored.

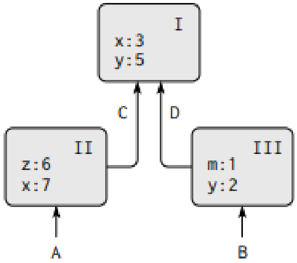
Programming that makes extensive use of assignment is known as *imperative* programming.

In the presence of assignment, a variable can no longer be considered to be merely a name for a value.
Rather, a variable must somehow designate a “place” in which values can be stored.

In our new model of evaluation, these places will be maintained in structures called *environments*.



- **An environment is a sequence of frames**.
  The diagram shows a simple environment structure consisting of three frames, labeled I, II, and III



- **An environment is a sequence of frames**.
  The diagram shows a simple environment structure consisting of three frames, labeled I, II, and III
- **Each frame also has a pointer to its enclosing environment, unless, the frame is considered to be global.**
  In the diagram, A, B, C, and D are pointers to environments. C and D point to the same environment.

- Each frame is a table (possibly empty) of bindings, which associate *variable names* with their *corresponding values*.

- Each frame is a table (possibly empty) of bindings, which associate *variable names* with their *corresponding values*.
- A single frame may contain at most one binding for any variable.

- Each frame is a table (possibly empty) of bindings, which associate *variable names* with their *corresponding values*.
- A single frame may contain at most one binding for any variable.
- The value of a variable with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable.
  $\rightarrow$ For example, the variables z and x are bound in frame II, while x and y are bound in frame I.
  The value of x in environment D is 3.
  The value of x wrt environment B is also 3.

- Each frame is a table (possibly empty) of bindings, which associate *variable names* with their *corresponding values*.
- A single frame may contain at most one binding for any variable.
- The value of a variable with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable.
  $\rightarrow$ For example, the variables z and x are bound in frame II, while x and y are bound in frame I.
  The value of x in environment D is 3.
  The value of x wrt environment B is also 3.
- If no frame in the sequence specifies a binding for the variable, then the variable is said to be unbound in the environment.
  $\rightarrow$ For example, z is unbound wrt frame III.

- An expression acquires a meaning only wrt some environment in which it is evaluated.
  $\rightarrow$ For example, the interpretation of the expression (+ 1 1) depends on an understanding that one is operating in a context in which + is the symbol for addition.

  Thus, in our model of evaluation we will always speak of evaluating an expression with respect to some environment.

- An expression acquires a meaning only wrt some environment in which it is evaluated.
  $\rightarrow$ For example, the interpretation of the expression $(+\ 1\ 1)$ depends on an understanding that one is operating in a context in which $+$ is the symbol for addition.

  Thus, in our model of evaluation we will always speak of evaluating an expression with respect to some environment.

- An expression acquires a meaning only wrt some environment in which it is evaluated.
  $\rightarrow$ For example, the interpretation of the expression $(+\ 1\ 1)$ depends on an understanding that one is operating in a context in which $+$ is the symbol for addition.

  Thus, in our model of evaluation we will always speak of evaluating an expression with respect to some environment.

- To describe interactions with the interpreter, we will suppose that there is a **global environment**, consisting of a single frame (with no enclosing environment) that includes values for the symbols associated with the primitive procedures.

- An expression acquires a meaning only wrt some environment in which it is evaluated.
  → For example, the interpretation of the expression (+ 1 1) depends on an understanding that one is operating in a context in which + is the symbol for addition.

  Thus, in our model of evaluation we will always speak of evaluating an expression with respect to some environment.

- To describe interactions with the interpreter, we will suppose that there is a **global environment**, consisting of a single frame (with no enclosing environment) that includes values for the symbols associated with the primitive procedures.
  → For example, the idea that + is the symbol for addition is captured by saying that the symbol + is bound in the global environment to the primitive addition procedure.

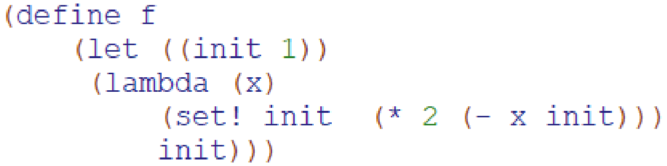
To evaluate a combination, do the following:

1. Evaluate the subexpressions of the combination.
2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).

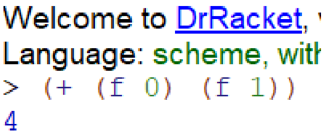
To evaluate a combination, do the following:

1. Evaluate the subexpressions of the combination.
2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).

Difference w.r.t substitution model:- The presence of assignment (set!) allows us to write expressions that will produce different values depending on the order in which the subexpressions in a combination are evaluated.

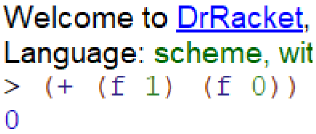


```
(define f
    (let ((init 1))
     (lambda (x)
         (set! init  (* 2 (- x init)))
         init)))
```



Welcome to DrRacket,
Language: scheme, wit

```
> (+ (f 0) (f 1))
4
```

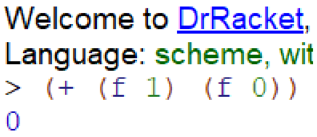


Welcome to DrRacket,
Language: scheme, wit

```
> (+ (f 1) (f 0))
0
```

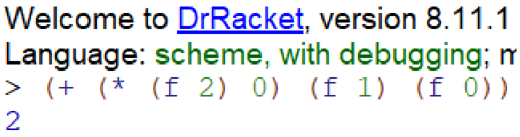
.
The order in which evaluation happens becomes relevant.

An compiler cannot optimize by avoiding computation of (f 2) as this would change the output.

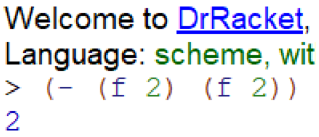


```
Welcome to DrRacket,
Language: scheme, wit
> (+ (f 1) (f 0))
0
```

.



```
Welcome to DrRacket, version 8.11.1
Language: scheme, with debugging; m
> (+ (* (f 2) 0) (f 1) (f 0))
2
```



```
Welcome to DrRacket,
Language: scheme, wit
> (- (f 2) (f 2))
2
```

Therefore, one should specify an evaluation order in step 1 (e.g., left to right or right to left)
or
at least be aware of the default order in which the evaluation is being done and assume that this default order will not be changed by the compiler.

Therefore, one should specify an evaluation order in step 1 (e.g., left to right or right to left)
or
at least be aware of the default order in which the evaluation is being done and assume that this default order will not be changed by the compiler.

These are implementation level details and a sophisticated compiler might optimize a program by varying the order in which subexpressions are evaluated.

The presence of set! restricts the freedom of the compiler from performing optimizations.

In the environment model of evaluation, a procedure is always a pair consisting of some **code** and a **pointer to an environment**.

Procedures are created by evaluating a λ-expression.

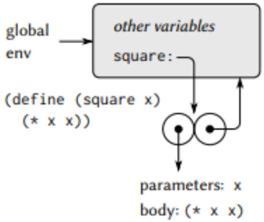
Evaluating a λ-expression produces a procedure

- whose code is obtained from the text of the λ-expression and
- whose environment is the environment in which the λ-expression was evaluated to produce the procedure.

Consider the procedure definition evaluated in the global environment.
(define square (λ(x) (* x x)))

1. Evaluate (λ(x) (* x x)) in the global environment.
2. Bind square to the resulting value in the global environment.

Now the environment (here global) has access to the evaluated code and refers to it using square and the procedure knows the environment in which it was evaluated.



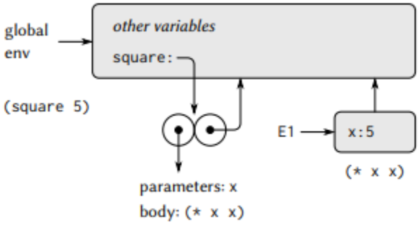
The procedure “object” is a pair whose code specifies that the procedure has one formal parameter, namely x, and a procedure body (* x x).

The environment part of the procedure is a pointer to the global environment, since that is the environment in which the λ-expression was evaluated to produce the procedure.

A new binding, which associates the procedure object with the symbol square, has been added to the global frame.

To apply a procedure to arguments, create a new environment containing a frame (here E1) that binds the parameters to the values of the arguments.

The enclosing environment of this frame is the environment specified by the procedure. Now, within this new environment, evaluate the procedure body.



Applying the procedure results in the creation of a new environment (E1), that begins with a frame in which x, the formal parameter for the procedure, is bound to the argument 5.

The pointer leading upward from this frame shows that the frame's enclosing environment is the global environment.

The global environment is chosen here, because this is the environment that is indicated as part of the square procedure object.

Within E1, we evaluate the body of the procedure, (* x x).
Since x in E1 is 5, the result is 25.

Behavior of set!

Evaluating the expression
(set! ⟨variable⟩ ⟨value⟩)
in some environment locates the binding of the variable in that environment and changes its binding to indicate the new value.

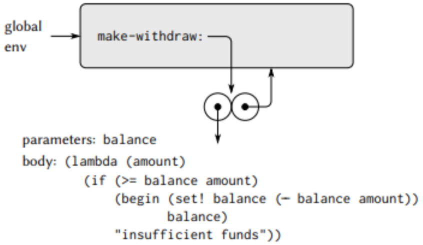
One finds the first frame in the environment that contains a binding for the variable and modifies that frame.

Behavior of set!

Evaluating the expression
(set! ⟨variable⟩ ⟨value⟩)
in some environment locates the binding of the variable in that environment and changes its binding to indicate the new value.

One finds the first frame in the environment that contains a binding for the variable and modifies that frame.

If the variable is unbound in the environment, then set! signals an error.

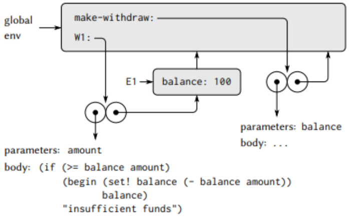


This is the result of defining the make-withdraw procedure in the global environment.

The body of the procedure is a λ-expression. This produces a procedure object that contains a pointer to the global environment.

(define W1 (make-withdraw 100))

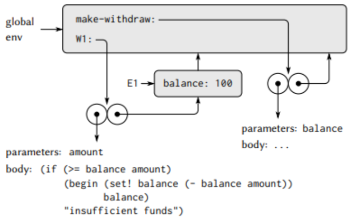
Applying an argument to 'make-withdraw', then defining it as W1.



1. Set up an env E1 in which the formal parameter balance is bound to the argument 100.
2. Evaluate the body of make-withdraw in E1.

(define W1 (make-withdraw 100))

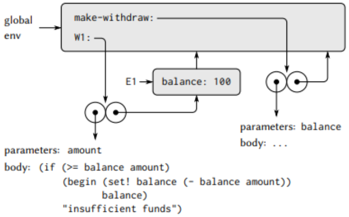
Applying an argument to 'make-withdraw', then defining it as W1.



1. Set up an env E1 in which the formal parameter balance is bound to the argument 100.
2. Evaluate the body of make-withdraw in E1.
3. This constructs a new procedure object, whose code is as specified by the lambda and whose env is E1.

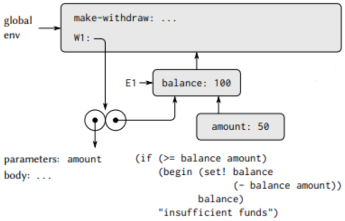
(define W1 (make-withdraw 100))

Applying an argument to 'make-withdraw', then defining it as W1.



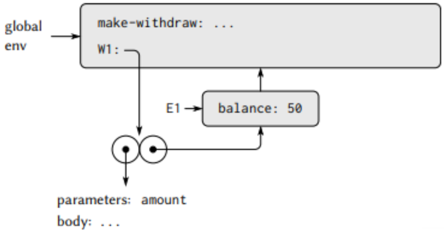
1. Set up an env E1 in which the formal parameter balance is bound to the argument 100.
2. Evaluate the body of make-withdraw in E1.
3. This constructs a new procedure object, whose code is as specified by the lambda and whose env is E1.
4. This new procedure object is bound to W1 in the global env, since the define itself is being evaluated in the global env.

(W1 50)



1. Construct a frame in which amount, the formal parameter of W1, is bound to the argument 50. This frame has E1 its enclosing environment, since E1 is the environment that is specified by the W1 procedure object.
2. Evaluate the body of the procedure.

After the procedure is evaluated for balance being 100 and amount being 50,



At the completion of the call to W1, balance is 50 (100-50), and the frame that contains balance is still pointed to by the procedure object W1.

The frame that binds 'amount' (in which we executed the code that changed balance) is no longer relevant, since the procedure call that constructed it has terminated, and there are no pointers to that frame from other parts of the environment.
So that frame exists for now, but since we have exited that frame and that frame is no longer reachable, we have not shown it in the diagram.

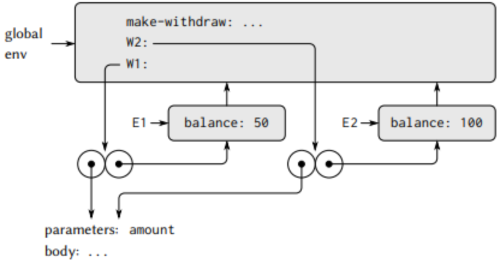
The next time W1 is called, this will build a new frame that binds **amount** and whose enclosing environment is E1.

The frame that binds 'amount' (in which we executed the code that changed balance) is no longer relevant, since the procedure call that constructed it has terminated, and there are no pointers to that frame from other parts of the environment.
So that frame exists for now, but since we have exited that frame and that frame is no longer reachable, we have not shown it in the diagram.

The next time W1 is called, this will build a new frame that binds **amount** and whose enclosing environment is E1.

We see that **E1 serves as the "place" that holds the local state variable for the procedure object W1.**

(define W2 (make-withdraw 100))

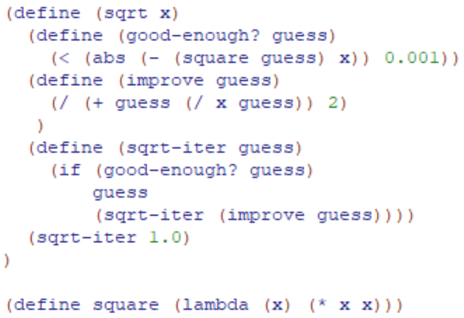


Similar to W1, W2 is a procedure object, that is, a pair with some code and an environment.
The environment E2 for W2 was created by the call to make-withdraw.
It contains a frame with a local binding for its own balance.

W1 and W2 have the same code: the code specified by the λ-expression in the body of make-withdraw.

We see here why W1 and W2 behave as independent objects.
Calls to W1 reference the state variable balance stored in E1, whereas
calls to W2 reference the state variable balance stored in E2.

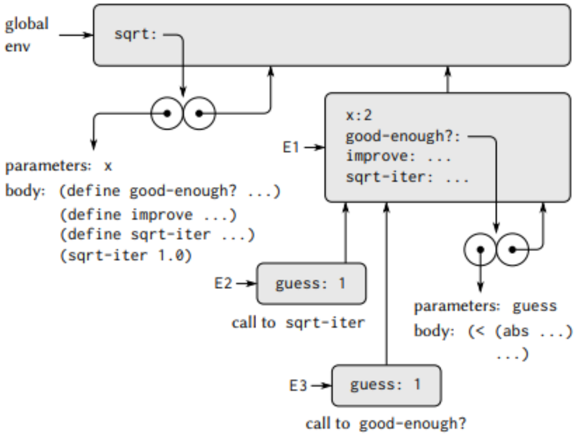
Thus, changes to the local state of one object do not affect the other object.



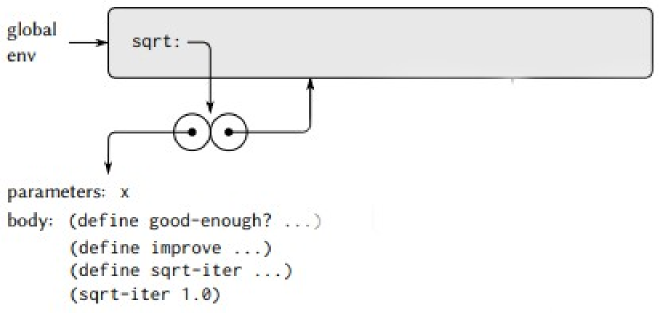
```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (/ (+ guess (/ x guess)) 2)
   )
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0)
)

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

The sqrt procedure has several internal procedures. We can use the environment model to see why these internal definitions behave as desired.



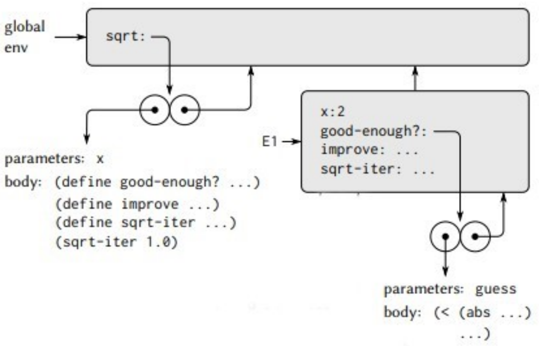
The figure shows the point in the evaluation of the expression (sqrt 2)



sqrt is a symbol in the global environment that is bound to a procedure object whose associated environment is the global environment.

When sqrt was called with 2 as the parameter, a new environment E1 was formed, subordinate to the global environment, in which the parameter x is bound to 2.

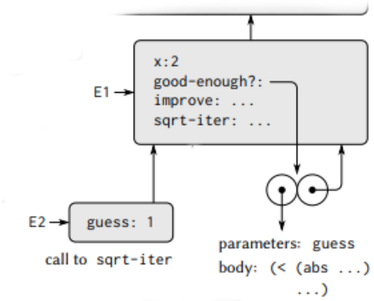
The body of sqrt was then evaluated in E1.
So everything in the E1 and its subordinate frames came into existence after 2 was applied to sqrt.



While evaluating the body of E1, we first defined the procedure good-enough? in the environment E1. In other words, the symbol good-enough? was added to the first frame of E1, bound to a procedure object whose associated environment is E1.

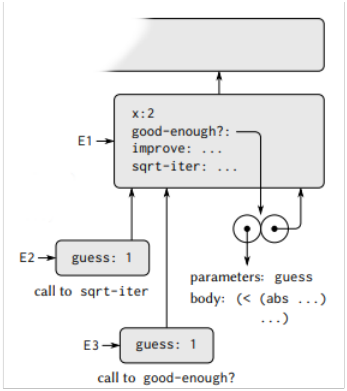
Similarly, improve and sqrt-iter were defined as procedures in E1.

After the local procedures were defined, the expression (sqrt-iter 1.0) was evaluated, still in environment E1. So the procedure object bound to sqrt-iter in E1 was called with 1 as an argument.



This created an environment E2 in which guess, the parameter of sqrt-iter, is bound to 1.

sqrt-iter in turn called good-enough? with the value of guess (from E2) as the argument for good-enough?.



This set up another environment, E3, in which guess (the parameter of good-enough?) is bound to 1.

Although sqrt-iter and good-enough? both have a parameter named guess, these are two distinct local variables located in different frames.

Also, E2 and E3 both have E1 as their enclosing environment, because the sqrt-iter and good-enough? procedures both have E1 as their environment part.

This implies that the symbol x which appears in the body of good-enough? will reference the binding of x that appears in E1.

The environment model thus explains the two key properties that make local procedure definitions a useful technique for modularizing programs:

The environment model thus explains the two key properties that make local procedure definitions a useful technique for modularizing programs:

- The names of the local procedures do not interfere with names external to the enclosing procedure, because the local procedure names will be bound in the frame that the procedure creates when it is run, rather than being bound in the global environment.

The environment model thus explains the two key properties that make local procedure definitions a useful technique for modularizing programs:

- The names of the local procedures do not interfere with names external to the enclosing procedure, because the local procedure names will be bound in the frame that the procedure creates when it is run, rather than being bound in the global environment.

- The local procedures can access the arguments of the enclosing procedure, simply by using parameter names as free variables. This is because the body of the local procedure is evaluated in an environment that is subordinate to the evaluation environment for the enclosing procedure.