An assignment statement is used to assign the value on the right to the variable on the left which has already been created.

An assignment statement is used to assign the value on the right to the variable on the left which has already been created.

In languages like C++ or Java those assignment statements are quite common and probably covered in the first few introductory sessions.

```
int myNum = 15;
myNum = 20; // myNum takes a new value of 20
```

Surprisingly, we were not required to use such features to solve any computational problem till now.

Till now whatever programming concepts that we have looked at, we tested those concepts in a language which did not require an assigning a new value to a pre-existing variable (except while looking at closures, which was an unrelated/standalone topic w.r.t. the rest of the topics).

Till now whatever programming concepts that we have looked at, we tested those concepts in a language which did not require an assigning a new value to a pre-existing variable (except while looking at closures, which was an unrelated/standalone topic w.r.t. the rest of the topics).

In case of closures, we were trying to prove the existence of something other than the code of the procedure being made use of, during the execution of the procedure. It illustrated something which persists even after the code has been executed and which impacts the execution of the same procedure later on.

Till now whatever programming concepts that we have looked at, we tested those concepts in a language which did not require an assigning a new value to a pre-existing variable (except while looking at closures, which was an unrelated/standalone topic w.r.t. the rest of the topics).

In case of closures, we were trying to prove the existence of something other than the code of the procedure being made use of, during the execution of the procedure. It illustrated something which persists even after the code has been executed and which impacts the execution of the same procedure later on.

**Why create something which persists even after the execution is complete?**

set! allows you to change the value of an operand, however it does not allow you to create a new operand and assign value to it, like define.

We need to answer some important questions?

- **Why add assignment (aka set!) as a feature to the language?**
  Is there some task which could not be accomplished due to lack of the assignment statements?

set! allows you to change the value of an operand, however it does not allow you to create a new operand and assign value to it, like define.

We need to answer some important questions?

- **Why add assignment (aka set!) as a feature to the language?**
  Is there some task which could not be accomplished due to lack of the assignment statements?
- **Does adding the assignment statement to a language create some negative side effects?**

Till now, we have been using functional programs.

Functional programs encode mathematical truths.

In the first few lectures, we were introduced to the *substitution model* of computation. This model allowed us to write functional programs and has served us well up until now.

Till now, we have been using functional programs.

Functional programs encode mathematical truths.

In the first few lectures, we were introduced to the *substitution model* of computation. This model allowed us to write functional programs and has served us well up until now.

However, while understanding closures, we looked at an operation called set!

```
(define foo-counter
                (let ((count 1))
                   (lambda ()
                     (set! count (+ count 1))
                       count)))
```

The use of set! in this context showed that procedures in scheme have an associated environment.

This was the only instance in the past several lectures, where we encountered a procedure that had explored the idea that *"running a procedure with the same operands could produce different results"*.

Allowing assignment in a language (for example, using set! in scheme breaks the substitution model.

The use of set! in this context showed that procedures in scheme have an associated environment.

This was the only instance in the past several lectures, where we encountered a procedure that had explored the idea that *"running a procedure with the same operands could produce different results"*.

Allowing assignment in a language (for example, using set! in scheme breaks the substitution model.

For example, in case of (foo-counter), one cannot substitute the value of count to be 1 and expect (foo-counter) to return 2 everytime.

Substitution model captures mathematical truths.
For example, a procedure computing factorial was equivalent to the mathematical function of factorial.

```
(define (factorial_1 n)
  (define (iter counter product)
    (if (> counter n)
        product
        (iter (+ counter 1) (* counter product))))
  (iter 1 1))
```

→ Mathematical truths do not change over time.

→ Mathematical truths do not have the concept of time.

One could also write the factorial procedure using set!.

```
(define (factorial_2 n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! counter (+ counter 1))
                 (set! product (* counter product))
                 (iter))))
    (iter)))
```

One could also write the factorial procedure using set!.

```
(define (factorial_2 n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! counter (+ counter 1))
                 (set! product (* counter product))
                 (iter))))
    (iter)))
```

This is an incorrect program for factorial. To correct the program, one needs to swap the order of the two set! statements.

The first factorial procedure makes a seperate copy of the body of the procedure each time. Whereas in the second factorial procedure, the procedure is not copied, only some of the variables keep changing. The second factorial procedure, requires us to take care of the order in which the two variables are updated.

An operation, function or expression is said to have a side effect if it modifies some state variable value(s) outside its local environment, which is to say if it has any observable effect other than its primary effect of returning a value to the invoker of the operation.

Examples

1. modifying a non-local variable,
2. modifying a static local variable (like in case of let),
3. modifying a mutable argument passed by reference,
4. performing I/O or calling other functions with side-effects.

In the presence of side effects, a program's behaviour may depend on history; that is, the order of evaluation matters.

In a nutshell, a function is said to have side effects when it depends on or modifies a state outside its scope.

Introduction of set! as a feature in the programming language has several side-effects which were previously not observed.

```
(define count 0)

(define (demo x)
 (set! count (+ 1 count))
      (+ x count))
```

The procedure has a different output for the same input.
This behaviour makes us realises that procedures are not equivalent to mathematical functions, if the language has set! as a tool.

```
> (demo 1)
2
> (demo 1)
3
> (demo 1)
4
```

Till now we have not encountered situations where the order of computation has affected the outcome.

However using set! one can create situations where different order of evaluation for an expression can produce different results.

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


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

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D1 (make-decrementer 100))
(define D2 (make-decrementer 100))

(D1 10)
(D2 10)

; D1 and D2 could be considered same,
; because any value of x, (D1 x) = (D2 x)
```

Are D1 and D2 the same?

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D1 (make-decrementer 100))
(define D2 (make-decrementer 100))

(D1 10)
(D2 10)

; D1 and D2 could be considered same,
; because any value of x, (D1 x) = (D2 x)
```

Are D1 and D2 the same?

An acceptable answer is yes, because D1 and D2 have the same computational behavior—each is a procedure that subtracts its input from 90.
In fact, D1 could be substituted for D2 in any computation without changing the result.

```
; Same procedure using set!
(define (make-decrementer-2 balance)
  (lambda (amount)
    (set! balance (- balance amount)) balance))

; D3 and D4 could be considered same at this point
(define D3 (make-decrementer-2 100))
(define D4 (make-decrementer-2 100))

(D3 10)
; Now D3 and D4 would be considered different,
; because (D3 x) \= (D4 x) for several values of x

(D3 20)
(D4 20)
```

Are D3 and D4 the same?

```
; Same procedure using set!
(define (make-decrementer-2 balance)
  (lambda (amount)
    (set! balance (- balance amount)) balance))

; D3 and D4 could be considered same at this point
(define D3 (make-decrementer-2 100))
(define D4 (make-decrementer-2 100))

(D3 10)
; Now D3 and D4 would be considered different,
; because (D3 x) \= (D4 x) for several values of x

(D3 20)
(D4 20)
```

Are D3 and D4 the same?

At point of creation they could have been considered same, but after (D3 10) instruction, they were clearly different entities, which could not be substituted for each other, without loss of meaning.

A language that supports the concept that "equals can be substituted for equals" in an expression without changing the value of the expression is said to be **referentially transparent**.

Referential transparency is violated when we include set! in our computer language.

This makes it tricky to determine when we can simplify expressions by substituting equivalent expressions.

Reasoning about programs that use assignment becomes drastically more difficult.

Once we forgo referential transparency, the notion of what it means for computational objects to be "the same" becomes difficult to capture in a formal way.

In general, we can determine that two apparently identical objects are indeed "the same one" only by modifying one object and then observing whether the other object has changed in the same way.

For example, in certain languages one can assign a memory location to a variable $x$ and then make $y$ an alias of $x$. If one is not sure whether $x$ and $y$ are referring to two different entities with same value or the same entity, then one can modify $x$'s value and observe whether $y$ has changed or not.

```
(define a 5)
(define b a)

(display "Values of a and b\n")
a b                                  Values of a and b
                                     5
;After modifying a                   5
(set! a 10)                          Values of a and b
(display "Values of a and b\n")      10
a b                                  5
```

Before adding set! to the language, we cannot make the distinction between *a* and *b*, by simply comparing *a* and *b*. If *a* and *b* can never be modified, then whether *a* and *b* are referring to the same entity or referring to two different entities having the same value would not be of any significance.

After adding set! to the language, we gain the ability to figure out whether *a* and *b* are two entities or not, as changing one entity did not change the other entity.

But how can we tell if a single object has "changed" other than by observing the "same" object twice and seeing whether some property of the object differs from one observation to the next?

Functional programming captures mathematical truths. In functional programming, the question "Is a equal to b?", makes perfect sense. However, if we are referring to the same object, the question would be "Is a equal to a?", which may seem redundant as mathematics assumes this to be true.

Thus,
we cannot determine sameness (of two objects) without observing the effects of change and
we cannot determine "change" (of a single object) without some a priori notion of "sameness,"

Introducing set! as a feature creates a different notion for operands.

Without set!, the operands were being used in the mathematical sense like an alias for a value.

If one allows the notion of changing an entity's value, it introduces the notion that the entity has an existence independent of its value, say a location in the memory and the value stored at that location can change.

So far, allowing set! to be part of the language has created few issues. There may be other issues arising as well.

- One needs to be **careful about the order in which set! assignments are used**, as seen in the example of factorial.

- The language now **loses referential transparency**. It breaks the substitution model of computation, so if one decides to add set! to the language, one needs a new model of computation. Assumptions regarding sameness of an entity (Is $a \stackrel{?}{=} a$) are now quite unclear.

  Without set! the value of $a$ at time $t$ was same as the value of $a$ at time $t + 2$. Once the language has the ability to modify $a$, how does one compare value of $a$ at ti
  me $t$ with the value of $a$ at time $t + 2$?

- One realises that **adding set! to the language also removes some features of the language**.
  For example, one cannot assume that an expression which is commutative in the substitution model, would still remain commutative after the use of set!, as seen in the case of evaluation of (+ (f 0) (f 1)).

Thus, if one intends is to add set! to the langauge, one must have a strong reason for adding it to the model of computation.

One must be able to do something of great value, which was previously not possible, such that the advantages far outweigh the disadvantages.