Till now, we had encountered recursively defined procedures and using streams we have created recursively defined data objects.

But that does not come as a surprise at this point, because you now intuitively realize that there is no difference between procedures and data.

We have created two classes of procedures:

- ordinary procedures and
- procedures that take delayed arguments.

In general, creating separate classes of procedures forces us to create separate classes of higher-order procedures as well.

One way to avoid the need for two different classes of procedures is to make all procedures take delayed arguments by default.

We could adopt a model of evaluation in which all arguments to procedures are automatically delayed and arguments are forced only when they are actually needed.

We could adopt a model of evaluation in which all arguments to procedures are automatically delayed and arguments are forced only when they are actually needed.

This would transform our language to use **normal-order evaluation**.

We could adopt a model of evaluation in which all arguments to procedures are automatically delayed and arguments are forced only when they are actually needed.

This would transform our language to use **normal-order evaluation**.

Converting to normal-order evaluation provides a uniform and elegant way to simplify the use of delayed evaluation, and this would be a natural strategy to adopt if we were concerned only with stream processing.

We could modify the language so that every procedure is automatically has an implicit delay around its arguments.

Miranda is such a language which captures this idea.

We could modify the language so that every procedure is automatically has an implicit delay around its arguments.

Miranda is such a language which captures this idea.

In such a language, if procedure-1 uses procedure-2 which in turn uses procedure-3 and so on, then procedure-3 would pass a promise to procedure-2 which in turn would pass a promise to procedure-1 and so on.

At some point the memory used by the promises could be huge and one would face trouble expressing iterative procedures.

Can we completely avoid assignment using streams?

Can we completely avoid assignment using streams?

Unfortunately no.

The use of streams would cause the system to do away with the notion of time.

However, we do need this notion of time in our interaction with the worldly objects and thus we cannot do away with assignment (set!) even if set! has its flaws.

Let's look at a simple procedure and understand its execution with and without delay, or in other words without and with time.

```
(define x 0)

(define (id n)
  (set! x n) n)

(define y (+ (id 5) 1))
x
y
x
```

If this piece of code uses normal order evaluation
Then the answer for the two x would be 0 and 5 respectively.
Between the two instructions to display x, we only tried to display y which should not have changed x.

This could be an issue if one tries to debug pieces of code and tries to understand the sequence of changes.

Delay can make it challenging to look at a piece of code and understand its control flow.

Delay removes the notion of time and that can be a problem, since we as humans have a very in-grained notion of time and sequence which is hard to neglect while writing and understanding programs.

Let's look at a real world situation with and without delay.

Preparing a dosa.

Without delay

1. get the ingredients for the receipe
2. follow the instructions in a specified order.

With delay

One would get the ingredients only when one uses that ingredient in the execution of that recipe.

So, if one needs to prepare dosa, with delay, one would start the process of grinding and fermenting the dosa batter after placing the pan on the stove, when one actually needs the batter.

One particularly troublesome area arises when we wish to design interactive systems, especially ones that model interactions between independent entities.
Example:- For a bank account, the notion of time is absolutely essential.
If a joint bank account has initial balance 100000 and has the following sequence of withdrawals from Peter and Paul as $\{1,2,3,4,5,900,\ldots\}$ and $\{1000,2,3,4\ \ldots\}$

Both the withdrawal streams are delayed and thus their actual length is not known which could be possibly infinite.

1. If Peter's withdrawal sequence is infinite, then one cannot complete processing Peter's withdrawal sequence to start processing Paul's withdrawal sequence.

   Possible fix - One could introduce some sort of inter-leaving mechanism to process both their withdrawal requests.
   (Say process 3 requests from Peter followed by 5 requests from paul and loop)

2 What if Paul does not have withdrawal requests for some intermediate subsequence?

Should the bank account manager wait for Paul to come with withdrawal request or should it switch over to processing Peter's request?

2 What if Paul does not have withdrawal requests for some intermediate subsequence?

Should the bank account manager wait for Paul to come with withdrawal request or should it switch over to processing Peter's request?

If one asks the manager to process Peter's requests, then it would mean that the manager waited(time!) for Paul's stream and didn't get anything so it switched to the other withdrawal stream.

One might also try to check if Peter/Paul's withdrawal stream is finite or not and then decide on the whether one should use interleaving or not, but then it would defeat the purpose of implementing delay and it would also not be possible to know if Peter's withdrawal is infinite or just really very very long.

Inadvertently, we again introduced the notion of time.

These two streams can be imagined as a single stream having two different mechanisms to produce alternate elements of the stream.

One way to understand the issue we face, would be visualize the bank account (after the interleave procedure) as a mathematical function which produces some output only after the single stream provides some sequence of input. But the two alternating procedures would face a problem if one of the streams stops providing elements.

The mathematical function of interleave is unable to interleave until it gets elements from both the streams.

One could however creates a clock-tick after which it keeps alternating between streams for input data, but that would then try to create the notion of time in functional programming which does not support that notion.

We introduced streams as a means to do away with set!, which as an unintended consequence ended up removing the notion of time from the system.

However when we are need to capture state using set!, we want the control over notion of time.

This is sort of a contradiction as to what ones want from streams.

Moral:- Delay (as a functional programming concept) is not able to capture the notion of time which is a real world concept. We previously thought of it as a positive feature, but turns out it is not as good as it seems.

As far as anyone knows, mutability and delayed evaluation do not mix well in programming languages, and devising ways to deal with both of these at once is an active area of research.

When we avoid set! and use streams, we are working in the functional programming paradigm which is suited for modelling mathematical functions.

Mathematical functions do not have the concept of time where as real world objects have a concept of time/sequential order, where certain things have to be done in a specified order.

These two different views of the world as possibly incompatible with each other and it is possible that one cannot avoid set! to model real world objects.

We can model the world as
a collection of separate, time-bound, interacting objects with state,
or

we can model the world as
a single, timeless, stateless unity.

Each view has powerful advantages, but neither view alone is completely satisfactory.

We can model the world as
a collection of separate, time-bound, interacting objects with state,
or

we can model the world as
a single, timeless, stateless unity.

Each view has powerful advantages, but neither view alone is completely satisfactory.

No one has figured out a way to unify these two views.
Unifying the object view with the functional view may have little to do with programming, but rather with something fundamentally different/unrelated.
For example, the way one studies quantum particles is different from the way one views/studies stars and one may not be able to unify the two views of the universe.