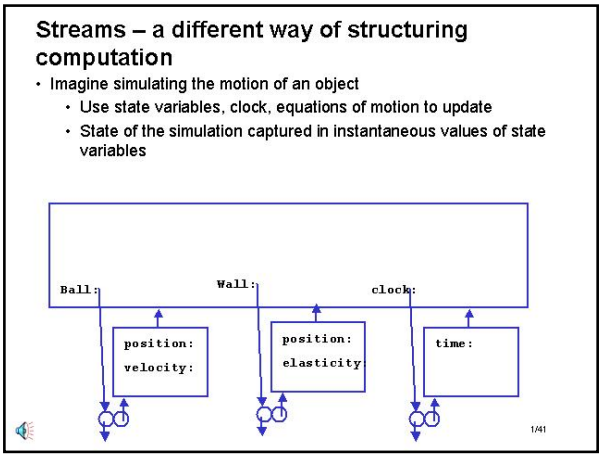
6.001 Notes: Section 17.5

Slide 17.5.1

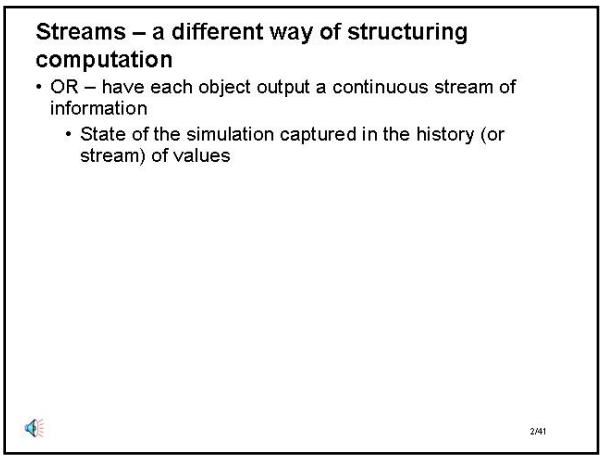


Now, let's look at one example in which changing the evaluation model allows us to explore a very different kind of computational problem. Our goal is to show how a small change in the evaluator, basically our lazy evaluator, can let us have a very different way of thinking about programs and programming.

Imagine that I want to simulate the motion of an object in a complex environment. A simple case might be a tennis ball that I throw against a set of walls. I would like to simulate how the ball would bounce against those obstacles and where it might end up. In our earlier approach, we might have chosen to model this using an object oriented system, which seems like a natural way of breaking this problem up into pieces. Under that view, we would have a different object to represent each different structure in our simulation world. For example, we might have an object that represented the ball, with some internal state that captured the properties of the ball. Similarly each wall would be an object, perhaps with different characteristics representing how objects bounce off them. And we might have a clock to synchronize interactions between the objects, leading to an object centered system very similar to what we saw in earlier lectures. In this way, each synchronization step would cause the objects to update their state, including detecting when, for example, two objects have collided so that the physics captured in each object would then govern changes in the state of the objects.

The thing to notice is that while this is a natural way of breaking up the system into units, the state of the simulation is basically captured in an instantaneous way. At any instant, we can determine the state of the overall system by the values of the state variables of each object. But we don't have a lot of information about how the system has been evolving.

Said a different way, by breaking up this system into units of this form, we are naturally focusing on the discrete objects within the system, not on the behavior of the system.



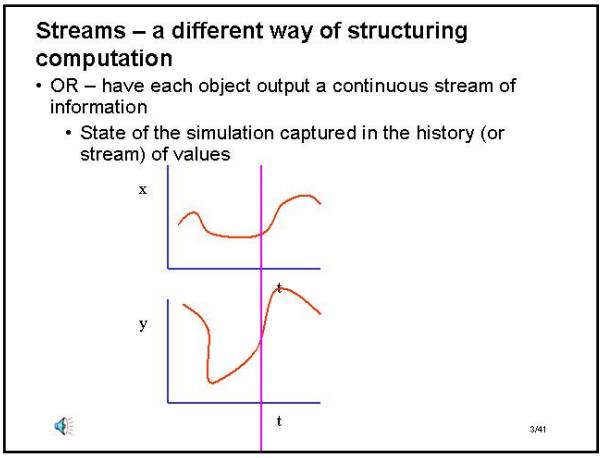
Slide 17.5.2

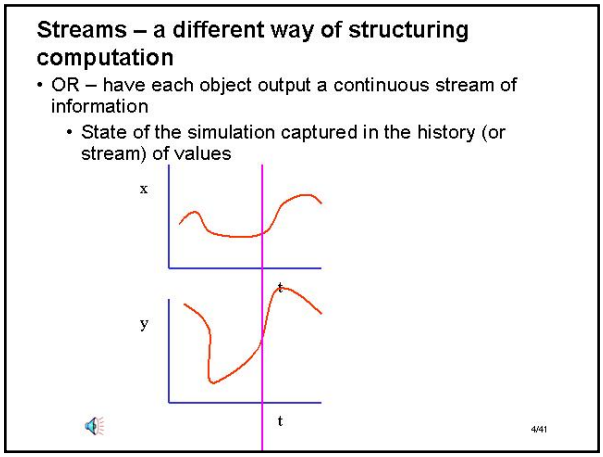
There is a very different way of thinking about such systems, however. Rather than having structures that capture state explicitly, I could think about systems in which the state information is only there in an implicit way. In my example of a tennis ball being thrown against a set of walls, imagine that while doing that action, I also include a set of cameras placed around the edges of the room. These cameras might record the movement of the ball, and thus can capture information about the state of the ball. In particular, imagine that this is happening in a continuous fashion. That is, there is a constant stream of information being spewed out that represents the **x** and **y** position (for example) of the ball as it moves around the room.

Slide 17.5.3

Under this view, my basic units now become the time series of values of the different variables that represent my system. In the earlier version, my basic units were the objects themselves: the ball, the walls, and the clock.

Now, I have changed my viewpoint. I have pulled out the state variables, and declared that my basic units are now the stream (or history) of values associated with each state variable. To capture the state of the system at any point, I simply take the values of all of those variables across the same point in time. But my units that I want to focus on are the actual stream of values, the time history of values associated with each state variable in my system.





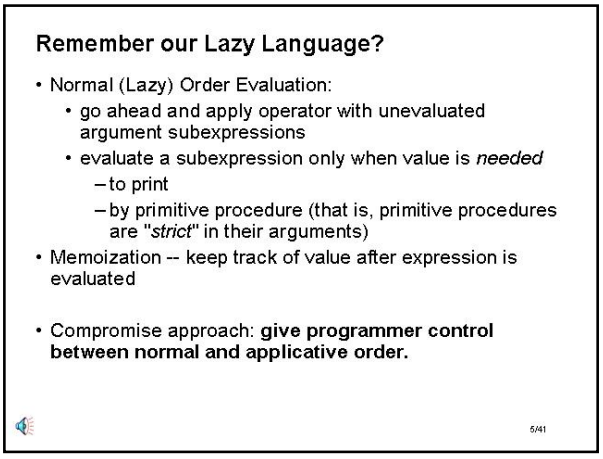
Slide 17.5.4

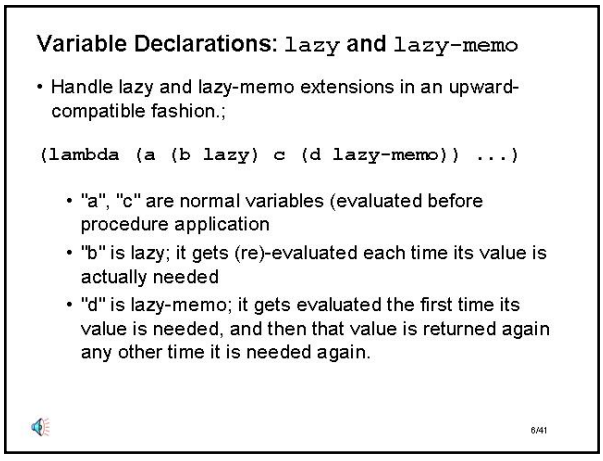
A key question, then, is how can I efficiently capture this kind of information? An obvious approach would be to just represent the history of values as a list. We could just glue new values of each variable onto the front of a list representing each such variable. While that is an appropriate way of conceptualizing the idea of capturing histories of values, we will see that when we move to complex systems, this becomes difficult to do in an efficient way. We would like to capture these histories with a minimum of computational effort, and for that we are going to return to the ideas we saw in the last lecture.

Slide 17.5.5

Now we just saw how to convert our standard, or applicative order, evaluator, into a normal order, or lazy, evaluator. I want to take that idea, and use it to change the way we think about programming, by showing how changing our viewpoint on evaluation coupled with this idea of capturing objects by their streams of values, gives us a very different way of programming.

The key ideas we are going to use are the notion of deferring evaluation of subexpressions until only when needed; and the idea of avoiding re-evaluation of the same subexpression, by memoizing it.

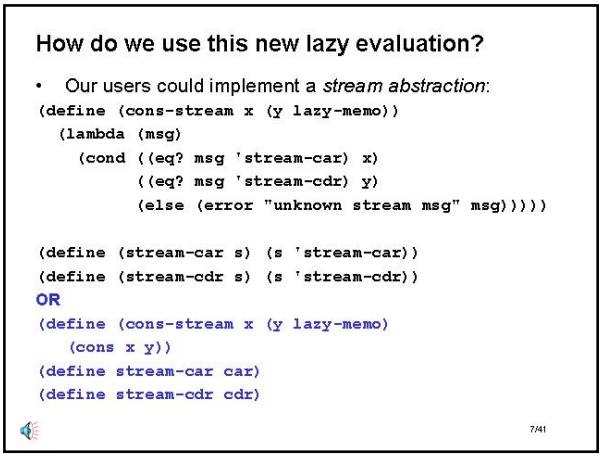




Slide 17.5.6

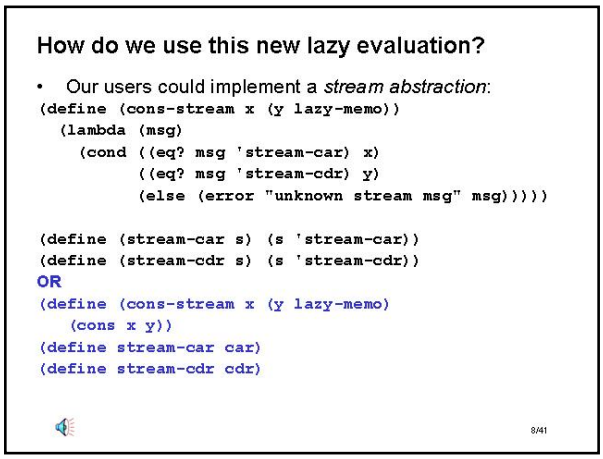
So we saw an evaluator in which the programmer could declare, when building a procedure, how to treat the different parameters. In this little example, a and c are normal variables, meaning that the expressions associated with them will be fully evaluated before we apply the procedure. Variable b we treat as lazy, meaning that we do not evaluate the associated expression at application but ather wait until the value is actually required within the body of the procedure (e.g. when a primitive procedure is applied to this expression). In this case, however, once the value associated with the expression has been used in that primitive application, it is discarded. Thus if the same expression is used somewhere else in the body, we will redo the work to compute it when it's value is needed. Variable d is also to be treated as lazy, but in this case, once we have obtained the actual value for the variable, we keep it around, and just use that value if any other part of the procedure body uses the same expression.

Slide 17.5.7



So how could we use this idea in our context? We could create a new data abstraction, called a **stream**. Here is one version of this abstraction. It has a constructor, `cons-stream` and two selectors, `stream-car` and `stream-cdr`. You can see by their names, that we expect them to behave a lot like lists, with one exception. The exception is that we want the second part of the stream to be lazy, or better yet memoized and lazy. This means that when I make a stream, I want to defer getting the value of the second part of the stream until I am actually required to.

Here we have represented this in a message passing system. We could also have done this using cons directly, although we have to be careful about ensuring that `cons` does not force the evaluation of the `cdr` part of the pairing until asked to by some other primitive operation.

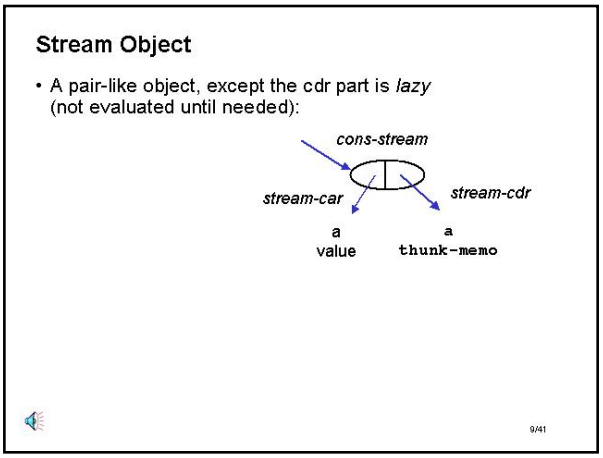


Slide 17.5.8

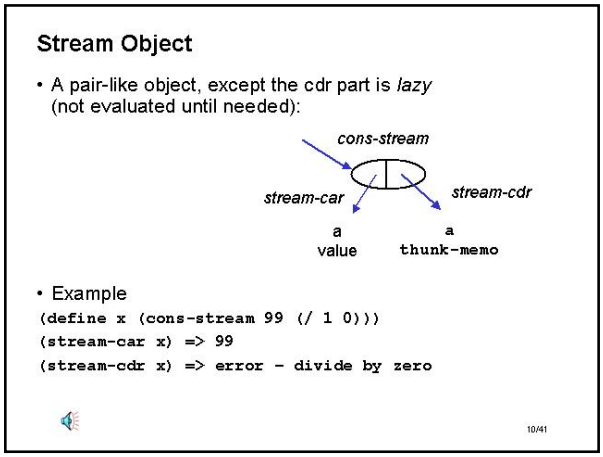
A key change is that now I have a way of gluing together a sequence of values in which only the **first** value is explicitly evaluated when I do the construction of the data object. The second part of this structure is **lazy**: it is a promise to get the value when asked for, but I don't do the work of computing the value at construction time.

Now, what does this do to our thinking about building sequences of values?

Slide 17.5.9



A stream object thus looks a lot like a pair, **except** that the `cdr` part of the stream is lazy. It is not evaluated until some procedure actually needs its value, but once we have worked out its value, we keep track of it for later reference. Think about what happens now. For example, if I ask `X` to have the value of a `cons-stream` of 99 and `(/ 1 0)` what will happen?



Slide 17.5.10

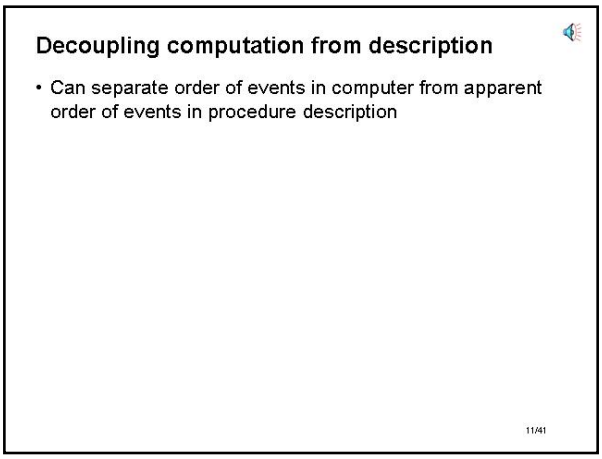
If I did this just using a normal `cons`, I would get an error, because `cons` would evaluate each of its arguments before constructing the pair, causing the division by zero to take place. In the case of `cons-stream`, we get a different behavior. `Cons-stream` will create an object with the value of the first argument as the first piece (99), but the second part is simply stored as a promise to compute a value when needed, and these two things are glued together. As a consequence, `x` is safely defined, and I get the first part of the stream without problem. It is only when I try to access the second part, using `stream-cdr`, that the evaluation of the deferred promise will take place, and I will see the error due to division by zero.

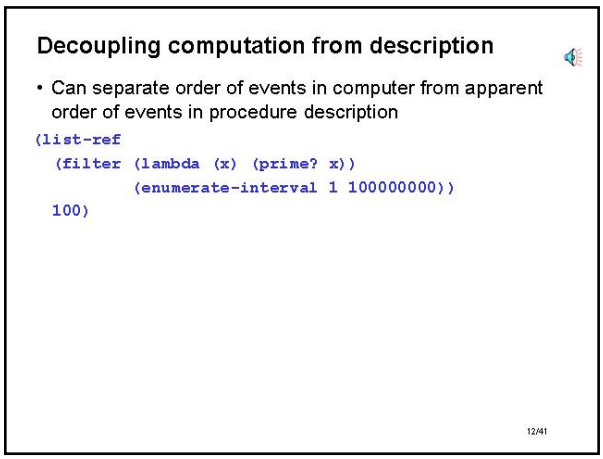
Thus, we see there is a difference between a stream object as a pair and a standard pair, in that the second part of a stream object is not evaluated until required. We are now going to build on that idea to see how we can create very different data structures and very different ways of thinking about computation.

Slide 17.5.11

This may seem like a very straightforward change. What I have in essence done is say: here is an alternative way of gluing to things together into pairs, specifically gluing together the value of the first thing with a promise to get the value of the second thing. It doesn't sound like a lot, but in fact it has a fundamental impact on how I can think about computation.

In particular, I can now **decouple the computation of values from the description of those values**, or said another way, I can **separate the order of events that occur inside of the computer from the apparent order of events as captured in the procedure description**. Let's look at an example.





Slide 17.5.12

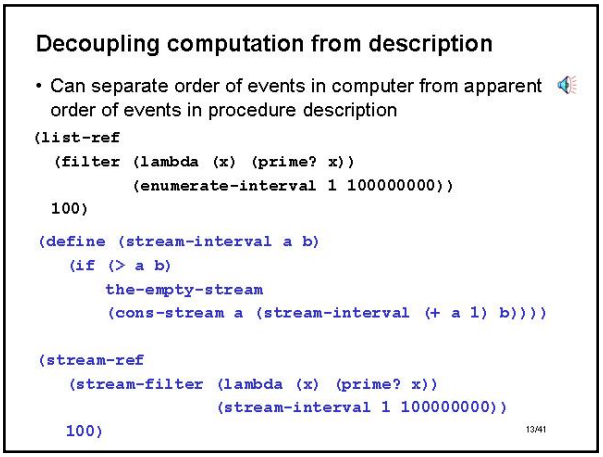
Suppose I want to find the value of the 100th prime. Here is the standard way of doing that, using lists. `Enumerate-interval` could generate a list of all the integers between 1 and 100,000,000. I could then filter that list, using a predicate that checks for primes (the details of the predicate are not important here). Given that new list of primes, I could then walk down the list to get the 100th element.

Notice what has happened here. I first had to generate a list 100,000,000 elements long, because I am not certain how many integers I will have to check before I find my answer. Thus I have done a lot of computation and I have chewed up a lot of memory creating a huge data structure. `Filter` then runs down that list and generates a new list, not quite as long as the original, but still a very large list, and involving a lot of computation. Finally, `list-ref` just walks down this new list, finds the 100th element, keeps it, and throws everything else away!

Using standard methods, I have to do all of the computation to get the value of an argument, before I can move on to the next stage of the computation. Thus, I have to generate an entire data structure before I can move on to

computations involving that data structure, even though much of the data structure may be irrelevant to my computation. In this example, that computation is wasted.

Slide 17.5.13

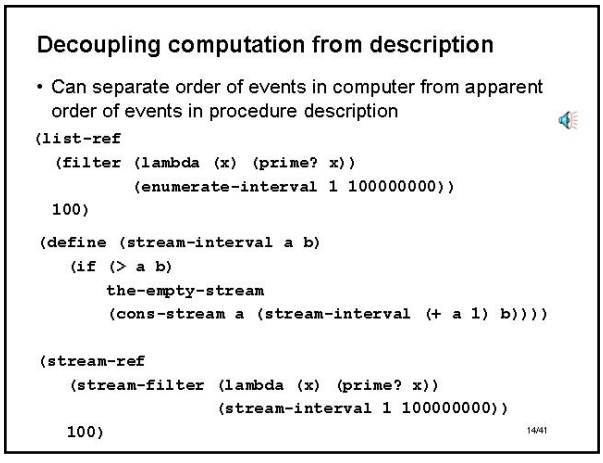


Suppose instead we change viewpoint. Rather than creating an entire list of integers before starting to look for primes, let's create a structure with the first integer that I need, and a promise to generate the rest of the integers when necessary. Rather than creating `enumerate-interval` as a very long list, I can use streams. Because of the lazy nature of streams, when I evaluate `(stream-interval a b)` I will get a data structure with the value of `a` as the first piece, and a **promise** to generate the rest of the interval from `(+ a 1)` to `b`, when required.

I can easily create a `stream-filter` that behaves like a filter on lists, but uses the constructor and selectors for streams instead of for pairs.

With that, notice what happens. Here, when I evaluate the last expression, `stream-interval` will generate a structure with the value `1` and a promise to generate the interval from 2 to 100,000,000. That can immediately be passed to `stream-filter` which will check to see if the first element is prime. Since it is not in this case, we will throw that value away, and ask for the next element in the stream. Remember that `stream-filter` will just walk down the stream the same way `filter` walked down a list. This will cause the computation to go back to `stream-interval` and ask it to generate the rest of its stream, the deferred second part. This will in turn ask `cons-stream` to generate the value `2` and **another promise** to generate the rest of the stream from here.

As a consequence, these two procedures will work in synchrony: `stream-interval` generating the next element in the stream, then passing that value plus a promise to `stream-filter` which will keep checking the value, and asking `stream-interval` for the next element until it finds one it likes. Thus, we will only generate as many elements in the stream as we need until `stream-ref` is able to extract the 100th prime.

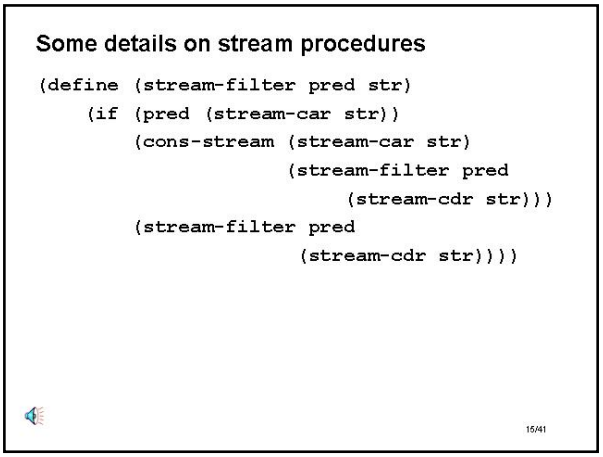


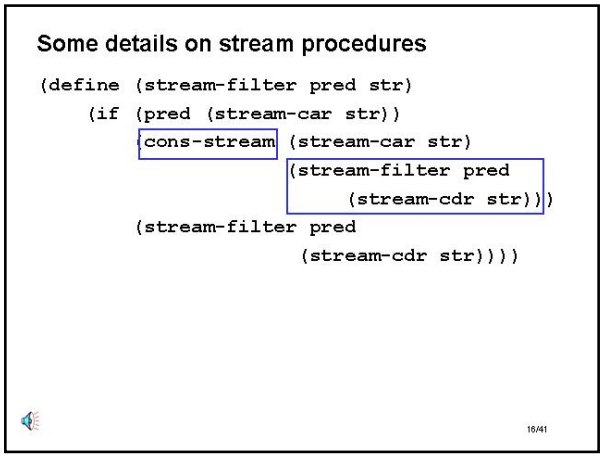
Slide 17.5.14

Note what this allows us to do. We can now think about the processing as if the **entire** set of values was available. We are thinking about how to deal with streams as if the entire sequence of values were there. But in fact, when we go to the computation, the laziness allows us to separate the order of events insides of the computer from the apparent order of events in the description of the process. Thus we get the efficiency of only computing what we need, while allowing us to think about things as if the entire sequence of values was available. To go back to our motivating example, we can build simulations in which we think about having the entire sequence of position values as if they were available, but we don't have to do all the computation needed to generate them in order to run the simulation.

Slide 17.5.15

To see how lazy evaluation gives us this behavior, let's look in a little more detail at this method. Here is a standard stream procedure, which looks exactly like our `filter` procedure for lists, the only difference is that we use stream abstractions in place of pair abstractions. But how does the lazy evaluation buried inside of `cons-stream` allow this procedure that looks just like a normal list procedure to have this different behavior, decoupling the order of evaluation within the machine from the apparent order described by the procedure?



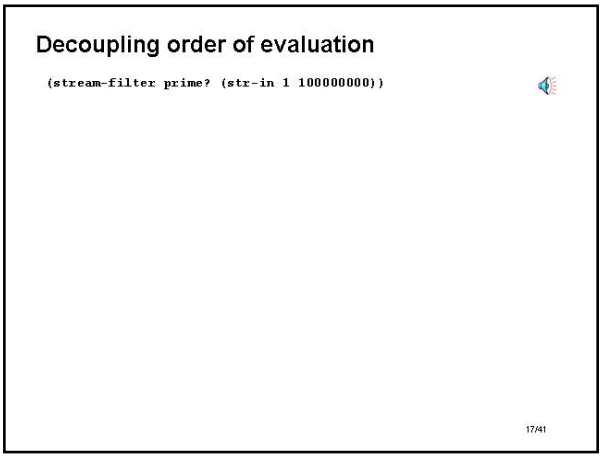


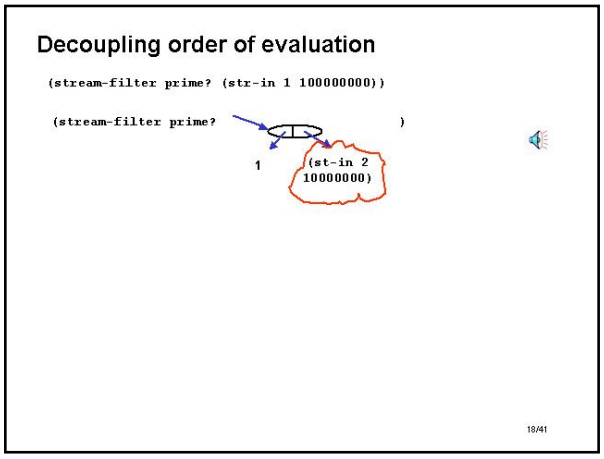
Slide 17.5.16

Indeed, a standard question might be "Why doesn't stream-filter end up generating all of the elements of the stream at once?" The answer is here. When we apply this procedure to a stream, it will recursively test each element in the stream, until it finds one that satisfies the predicate. At that stage, note what happens. We generate a stream with that element as the first element, and with a lazy (or delayed) promise to filter the rest of the stream when needed. Thus, we generate the first element of the new stream, and a lazy promise, not the entire stream.

Slide 17.5.17

Let's check it out on our simple example. Suppose we filter the stream of integers from 1 to 100,000,000, using the predicate `prime?`. Let's follow this computation, noting how lazy evaluation controls the order of evaluation of the parts of the data structure.



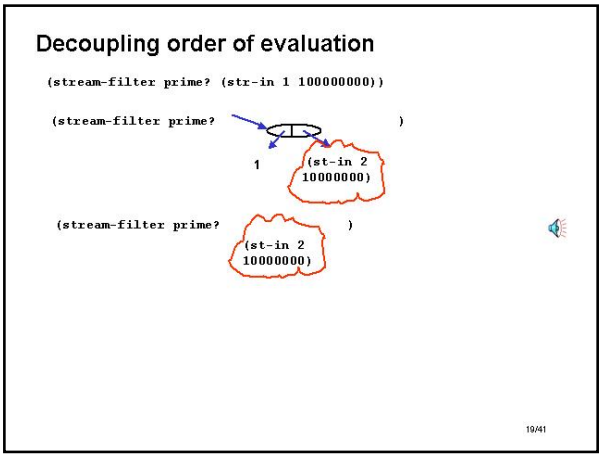


Slide 17.5.18

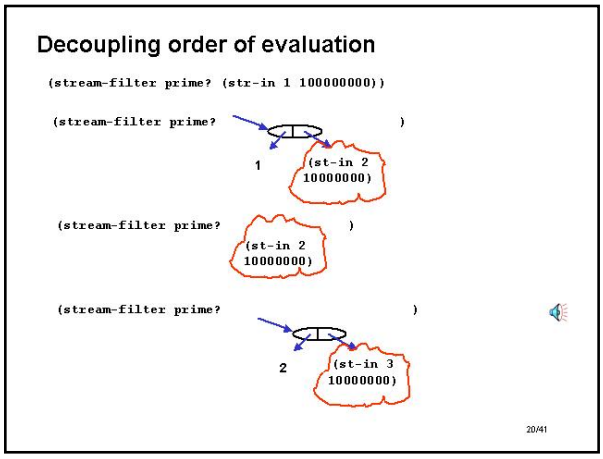
Since `stream-filter` is just a standard procedure, we need to get the values of its arguments. `Prime?` will simply point to some procedure. `(Stream-interval 1 100000000)` needs to be evaluated, but we know that `stream-interval` is defined in terms of `cons-stream`. So this returns one of these stream objects, which has the value of the first element, 1, already available, but simply has a promise (shown within that squiggly line) to get the value of the next argument, which is the stream interval from 2 to 100,000,000. At this stage, all that has been explicitly

computed is the first value. Everything else is just sitting around as a promise to do some work later.

Slide 17.5.19



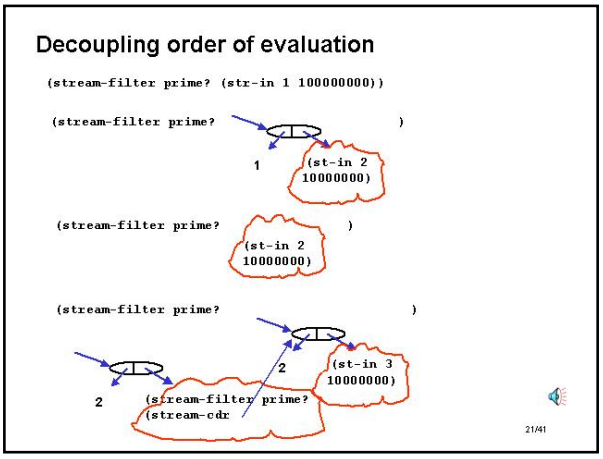
Having evaluated the two arguments to `stream-filter` we can now apply that procedure, i.e. evaluate its body. What does that do? It applies the predicate (the first argument) to the first value of the stream (the second argument). Since in this case 1 is not prime, `stream-filter` will return the value of the second clause of its `if` expression. This is a recursive call to `stream-filter` with the same predicate but now with the `stream-cdr` of the second argument. This means we should now force that promise, getting `stream-interval` from 2 to 100,000,000.



Slide 17.5.20

Here is where a potential confusion can arise. It might seem like evaluating the `stream-cdr` of this stream should cause all the remaining elements of the stream, everything from 2 to 100,000,000, to be evaluated. But remember that `stream-interval` says to do a `cons-stream` of the first argument onto a promise to get the rest of the values. So in fact it returns another stream object, with the next element in the sequence and another promise to generate the remaining elements. That is what will be supplied to `stream-filter`.

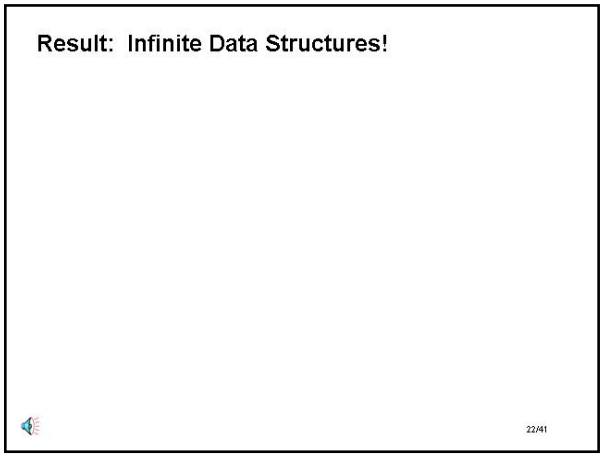
Slide 17.5.21



So now `stream-filter` can evaluate its body, testing with its predicate to see if the first element of this stream is a prime. It is, so it returns a `cons-stream` of the first element of the input stream, which is a 2, and a promise, and here the promise is to do a `stream-filter` on the remaining things. And notice what this `stream-filter` is. It is a promise to filter using `prime?` on the `stream-cdr` of the object we started with.

Thus we now have two delayed promises. We have a promise to do the filter, and inside of it is a promise to generate the rest of the initial stream. Thus we can see that we will only pull out values from the stream as we need them. If we ask for the next element in this stream, we would then force the evaluation of the `stream-filter` expression, which would force the evaluation of the `stream-interval` expression.

By hiding the lazy evaluation within the constructor, then building our procedures on top of that abstraction, we can easily enable the separation of the actual order of computation from the apparent order of computation.



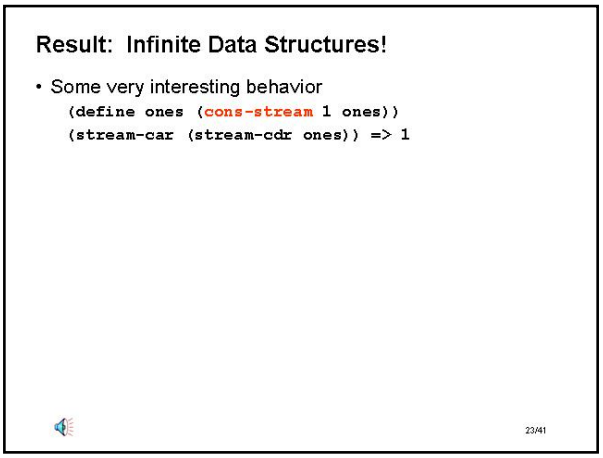
Slide 17.5.22

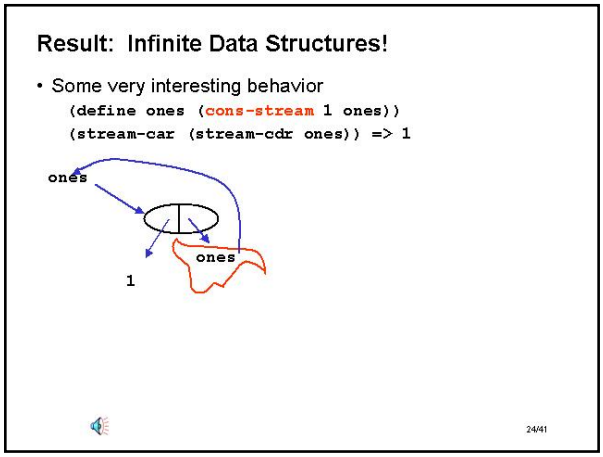
Now we see that if we create procedures that manipulate these stream objects, this new data structure, we never have to worry about how long the data structure actually is. We only get the next element in order as we ask for it. This raises an interesting question.

If we don't really care about how long the rest of the structure is, how long a structure could we make? The answer is: **infinitely long!** Actually, that is a slight mis-speaking, let's just say indefinitely long. We can now create data structures with arbitrary length that act as if they had infinite length, and this leads to some very interesting behavior in terms of how we think about processes. Let's look at an example.

Slide 17.5.23

Let's give the name `ones` to the structure we get by `cons-stream`ing the integer `1` onto `ones` itself. That sounds a bit weird. Note that if we ask for, say, the second element in this stream, we get out a `1`. Why is this happening?



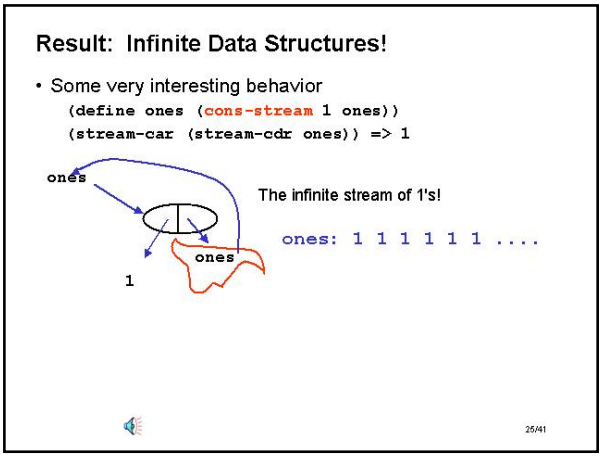


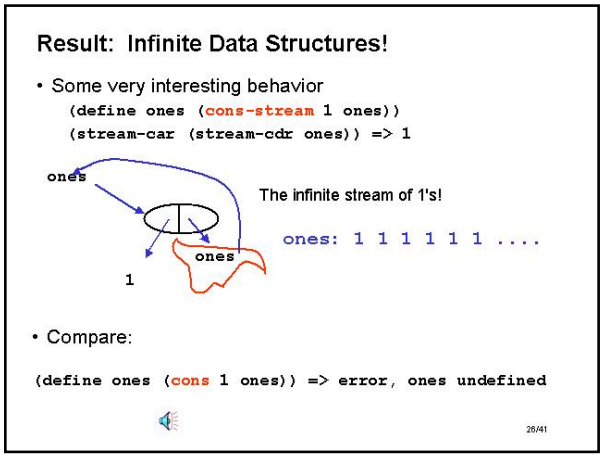
Slide 17.5.24

Well defining it this way says that the name `ones` refers to or points to a structure created by `cons-stream`ing the integer 1 onto a promise to get the value of the name `ones` when asked for it. And that says that only when we ask for the `stream-cdr` of this object will we evaluate this name, `ones`, which will simply point back to the binding we created for that name as part of the definition. As a consequence, this structure represents a very odd thing ...

Slide 17.5.25

... and infinite stream of 1's. No matter how many times I ask for the next element of this sequence, I will always get a 1. Thus I have a structure that I can conceptualize as representing an infinite set of things. Whenever I ask for some element in this sequence, it will always provide it to me, for any such element.



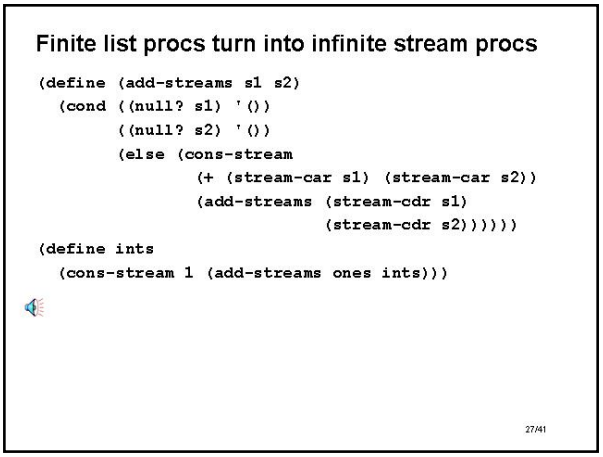


Slide 17.5.26

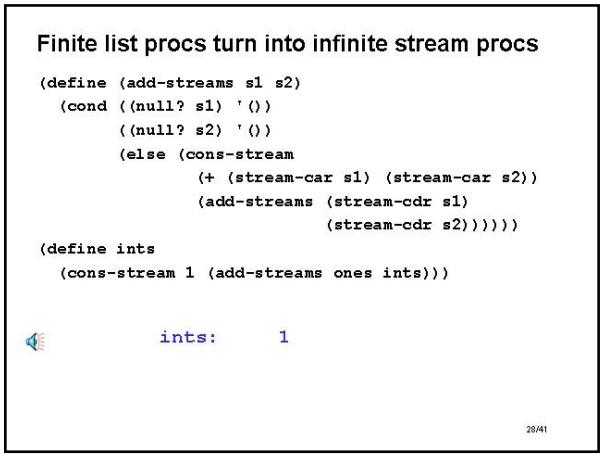
This may still seem odd, so let's think about the comparison to the standard evaluation model. Let's suppose that I tried to do this using `cons` in place of `cons-stream`. In this case, I get an error, because `cons` will evaluate both its arguments before constructing the data structure. Clearly I do not yet have a value for `ones`, so I can't glue it together with 1. With the lazy evaluation buried inside of streams, I can hold off on getting the value of this variable until I have completed the structure, which means the name `ones` will be bound to something when I go to get its value. Thus lazy evaluation provides the means necessary to enable creation of infinite data structures.

Slide 17.5.27

So what does this buy us? Well, this way of thinking about infinite data structures let's us think about creating procedures that operate as if the entire data structure were available to us. Thus procedures that typically apply to lists can be turned into procedures that handle infinite streams. Here is a procedure that adds together two streams, using the stream data abstraction. This should take two infinite streams as input and create an output stream in which the first element is computed by adding together the first elements of the input streams, and is glued onto a promise to add the remaining streams together when demanded.



Using this idea we can create an infinite data structure of all the integers. We simply `cons-stream` the first integer, which is 1, onto a promise to add together the stream of ones and the stream of integers. Let's check this out, since it seems a bit odd.



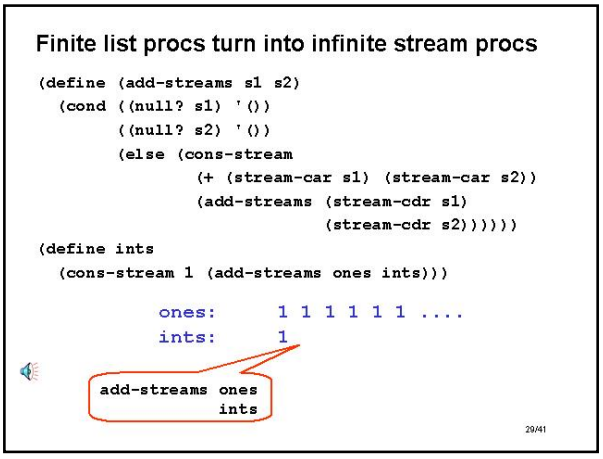
Slide 17.5.28

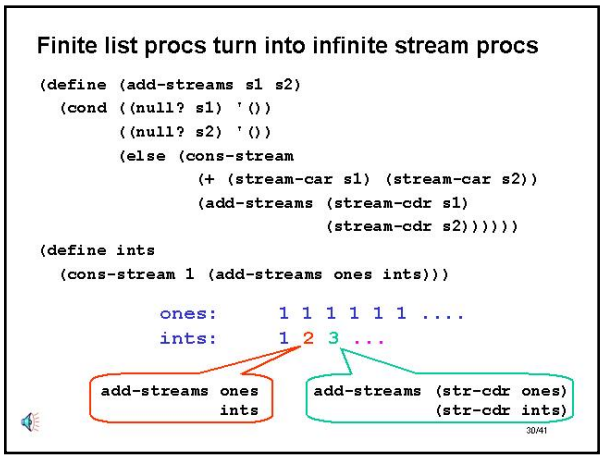
So what does `ints` or integers look like? Well we know that the first element will be a 1, because we `cons-stream`ed that onto a promise to get the rest of the integers. Okay, suppose we know ask for the second element of this stream...

Slide 17.5.29

... well the second element was a promise, a promise (because of the construction using lazy evaluation) to add together the stream of `ones` and the stream of `ints`. To get the second element we now need to evaluate that promise.

Notice that at this stage, both `ones` and `ints` are available, that is, have been bound to values in the environment. Each is a stream with an evaluated first element and a promise. And since the first elements are available, `add-streams` can thus add together the first elements of the two streams, and glue the sum together with a promise to add the remaining streams.

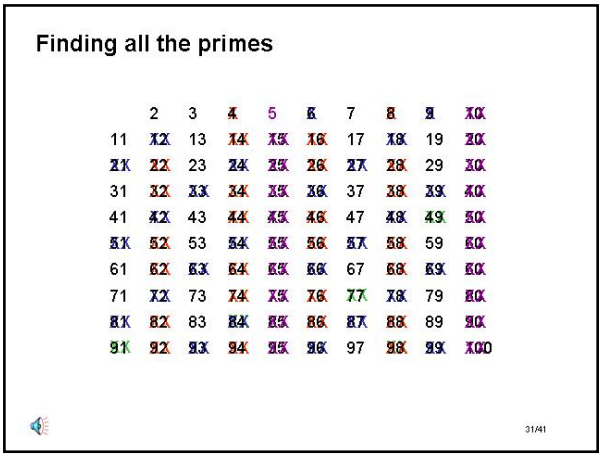




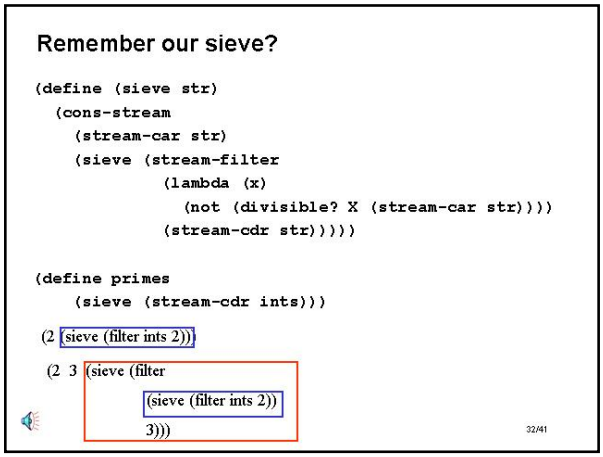
Slide 17.5.30

If we were to ask for the third element of `ints` we would then force this new promise. We would evaluate a promise to add together two streams: a promise to get the next element of `ones` and a promise to get the next element of `ints`. In this way, we can walk our way down `ints`. Whenever we ask for the next element in the sequence, `add-streams` will "tug" on the two input streams to generate the next element in each, add them, and generate a new promise to create the remainder of the stream.

Slide 17.5.31



Having the ability to create these infinite data structures, such as the infinite stream of ones, or the infinite stream of integers, let's us change our way of conceptualizing processes. In particular, remember the sieve that we saw many lectures ago. The Sieve of Erastosthenes said that to find all the primes, start with the integers beginning at 2, and do the following process. Take the next integer and include it as a prime. Then remove from the remaining set all integers that are divisible by this number. This creates a new set of integers. Now take the next one, include it as a prime, remove any elements in the rest of the sequence that are divisible by this element, and continue. Remember that when we did this using lists, we had to create a list of integers from 1 to some point, and then execute this process to get the first prime, plus generate a new list of all the remaining elements not divisible by 2, and so on. At each step, we had to generate a big list of values, only to throw much of it away. With streams, we can rethink this process.

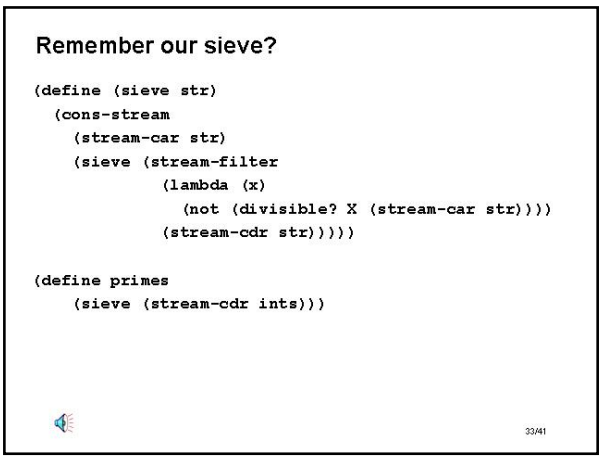


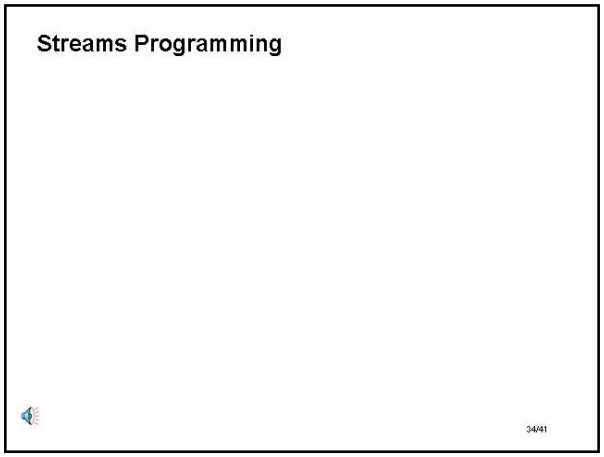
Slide 17.5.32

The idea of infinite data structures let's me create a sieve that is in fact much cleaner than the list version! As we said, in the list version, I would have to generate the entire list of integers, then filter that to generate another huge list, and so on.
On the other hand, with infinite data structures, I can generate the elements of the lists only as needed. Notice how I can do this. I can create a sieve that says: given an input stream, generate an output stream the first element of which will be the first element of the input stream. The rest of the stream will be a promise: a promise to take the rest of the input stream, filter it to remove anything divisible by the first element of the input stream, and then taking the `sieve` of that! That is nice, because that sieve will then, when asked to be evaluated, will generate the first element of that filtered stream, plus a promise to sieve again. This means that I will only pull out those elements from this sieve that I need, on demand. I can conceptualize the computation as if the entire sequence of elements is available, but only do the computation incrementally as needed.
Thus, I can define the primes as shown, and this structure will, when asked, compute as many primes as you request.

Slide 17.5.33

Notice, by the way, that this sieve definition does not include a base case! There is not test for the end of the input stream. But that is okay, because the input streams are infinite and we don't have to worry about reaching the end. The computation will simply keep generating more and more promises to get future elements as needed. Thus there is no base case, just the construction of the stream, element by element.



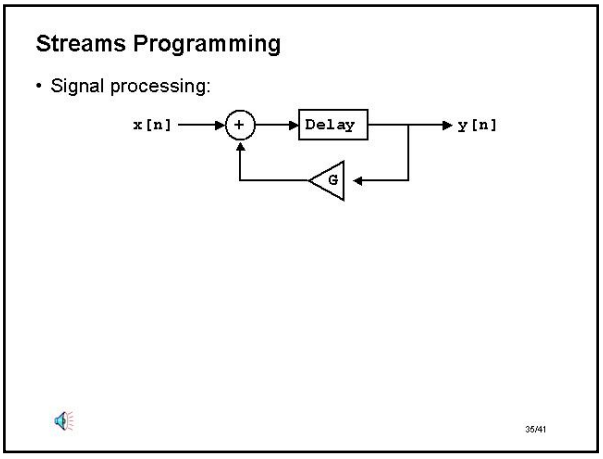


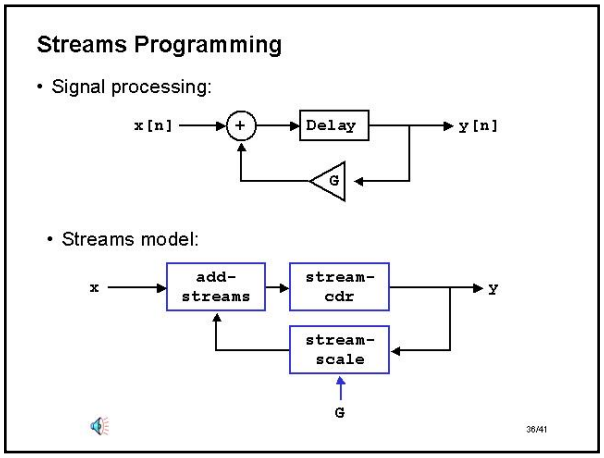
Slide 17.5.34

In fact this idea of thinking about these stream structures as infinite sequences of things that are generated as needed relates very nicely to a way of thinking about programming that is different than what we have seen so far. In fact, stream programming looks a lot like traditional signal processing. And what does that mean?

Slide 17.5.35

If we were to think about processing an audio signal, for example, the standard approach would be to consider some input signal `x` as a sequence of values. Then we would generate an output signal `y` by taking `x`, delaying that signal by some small amount, amplifying it by some process, then adding that value back into the input signal. This is a standard feedback loop, in which we get an output signal by processing some input signal. Using streams we can capture this idea of signal processing very cleanly.



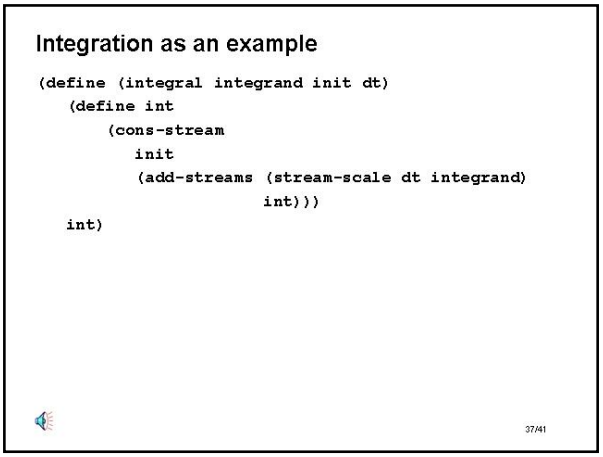


Slide 17.5.36

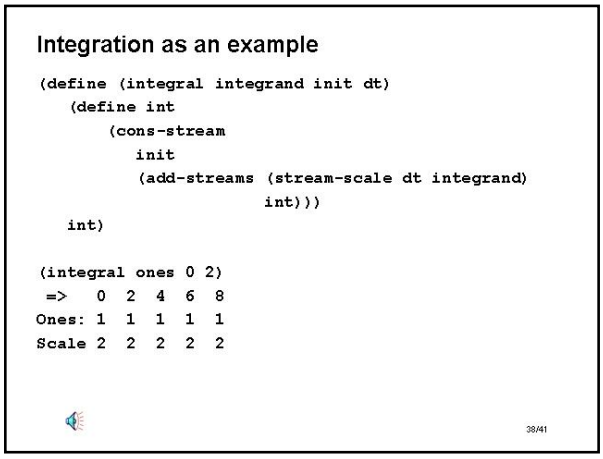
So we can capture this idea of a feedback loop very nicely in streams. If `x` is our input stream, we can generate an output stream `y` by the following. The first element of `x` will just become the first element of `y`. To get the next element of `y` we will use `stream-cdr` of `x` to get the second element of `x` (this is equivalent to delaying the stream), map that element through a procedure that amplifies the values, and adds that value to the value of `x` we just computed. This generates an infinite stream of output values based on an infinite stream of input values. Why would we like to have this process? It leads to a very common example of a feedback loop.

Slide 17.5.37

In fact exactly that idea of a feedback loop can be used to build an **integrator.** Suppose we had an integrand represented as a stream of values, that is, the height of a function at a sequence of points, thus tracing out a curve. We want to get the area under that curve, by integration. We could certainly generate a stream that corresponds to the sample values of the integrand. We start with some initial value (probably zero). To get the next value, that is the area under the curve to this point, we take the area computed so far (that is the value of the output stream), and add to it the next value of the input stream, multiplied by the spacing between the sample points (i.e. the area of the rectangle with width dx and height the value of the function, which is the approximation to the area of this section of the curve). Thus we can add up the area of the curve incrementally, generating the area obtained by adding in the next point to the area computed so far, all done incrementally on demand.



For example, we could use this to generate the integral of the stream of ones...

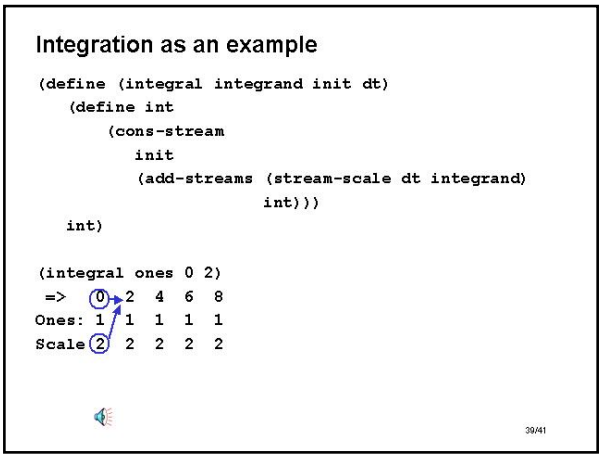


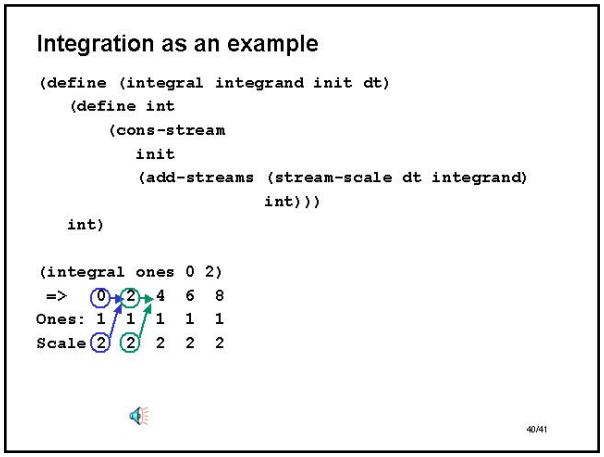
Slide 17.5.38

... if we take the integral of the stream of ones (i.e. the function with constant value 1), with a spacing of, say, 2, we get the following behavior. The first element of the returned stream is just the initial value, 0.

Slide 17.5.39

The next value in this integral stream is given by scaling the first element of the input stream and adding it to what we have added up so far. In terms of streams, we get the first value of ones, multiply it by two, and add it to the first of the integral, which we just computed. This results in the new value, 2, plus a promise to compute the next value in the sequence when needed.





Slide 17.5.40

If we ask `add-streams` to evaluate its next element, it will take the next element of the input (ones), scale it, and add it to the latest value of the output stream, and return that together with another promise for the next stage of the computation.

Slide 17.5.41

Key points are: having this idea of lazy evaluation supports the creation of structures that provide promises to create additional elements in a sequence when needed; and that those structures allow us conceptualize processes as if the entire sequence was available, focusing on the processing to be executed on the sequence, but have the actual evaluation occur just as needed.