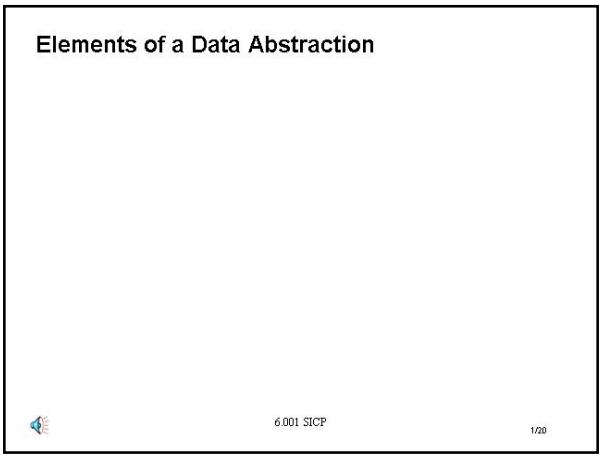
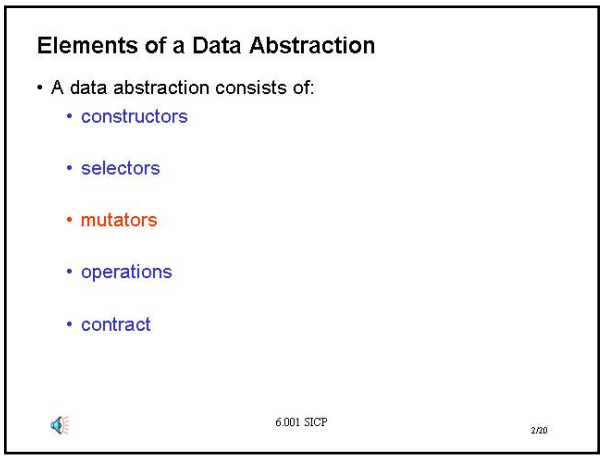
6.001 Notes: Section 11.1

Slide 11.1.1



For the past few lectures, we have been exploring the topic of data abstractions, and their role in modularizing complex systems. We have particularly looked at the relationship between data structures and the procedures that manipulate them.

Today we are going to add a new aspect to this topic, by introducing the idea of mutation. This is the idea of changing or altering an existing data structure, rather than simply making a copy of it. We are going to look at two examples of useful data structures, and show how while mutation carries some hazards with it; it also supports very efficient implementation of such structures.



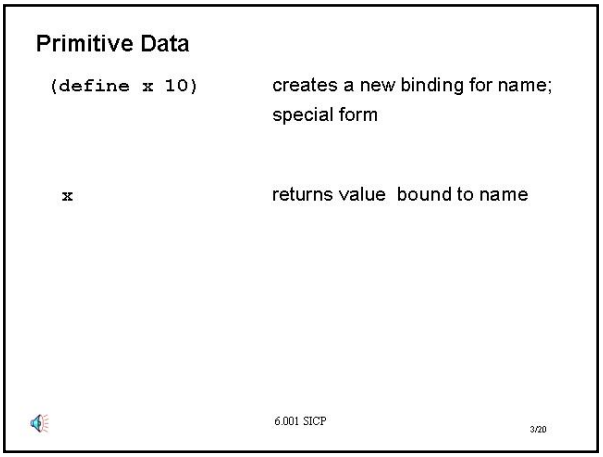
Slide 11.1.2

To set the stage for our exploration, let's first review what we know about data abstractions. First, we know that a data structure has a constructor, a way of gluing pieces together. Typically the definition of the constructor also involves a designation of the type contract of the constructor: what kind and how many objects are glued together.

Associated with the constructor are sets of accessors or selectors that get pieces of the object back out. These are governed by a contract with the constructor, designating the behavior by which pieces glued together can be pulled apart. And we typically had a set of operations that used the data structures without actually using the details of the implementation. The standard form is to use the selectors to get pieces of an existing object, manipulate these pieces to create new components, then reassemble these using the constructor into a new version of the data abstraction.

The new thing we are adding is a mutator. This is something that will **change** an existing data object, that is, go into the exist structure and alter pieces of that structure in place, rather than constructing a new version of the object with copies of the parts. This is the topic to which we now turn.

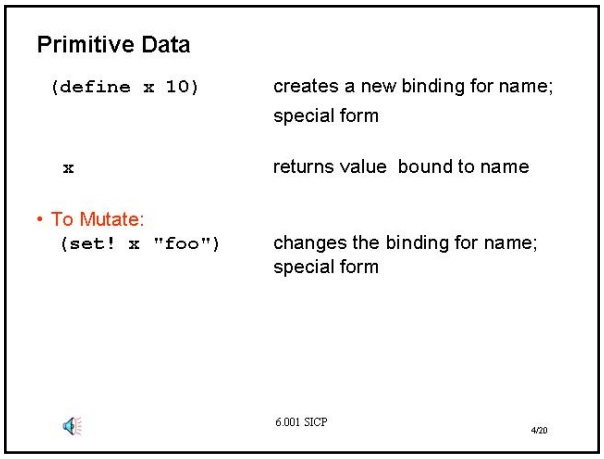
Slide 11.1.3



Let's start by looking at very simple data structures. For example, we can actually consider a **variable** or **name** to be a kind of data abstraction.

If we take this view, then our "constructor" in this case is the special form `define`. It binds the name (or symbol) given by the first argument to the value of the second expression. The key point is that we can consider this as a kind of primitive data abstraction that pairs up a name and a value.

The associated selector in this case is very simple. We just use the name itself, that is, when we evaluate the name, it looks up the value bound to it by the `define` expression, and returns that value, that is, it pulls apart the binding of name and value, and returns one component of that binding.



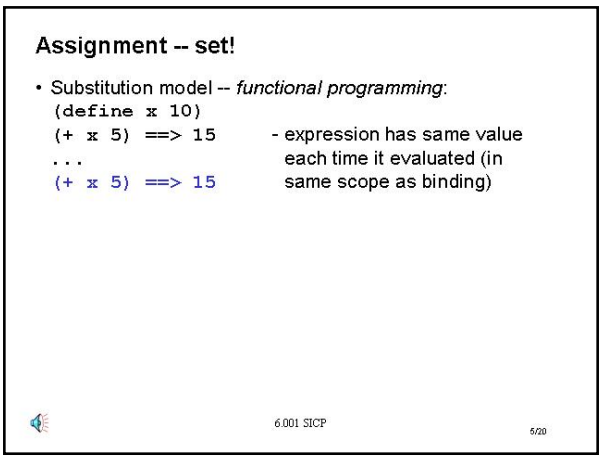
Slide 11.1.4

Now we can introduce the ability to mutate that variable, that is, to change the binding of the name and a value. Let's look first at the expression that accomplishes this, which is the `set!` expression shown.

This is also a special form, as it doesn't follow the normal rules for combinations. The rule for evaluation of this form is: take the second argument and evaluate it using standard rules; then take the first argument and just treat it as a symbol (i.e. don't evaluate it). Now, find the binding of that symbol and change it to take on the new value.

Note that this looks a lot like a `define` expression, but as we will see in a few lectures, there is a fundamental difference. For now, the distinction is that a `define` would create a new binding for the name, using up additional space, whereas `set!` changes an existing binding.

Slide 11.1.5

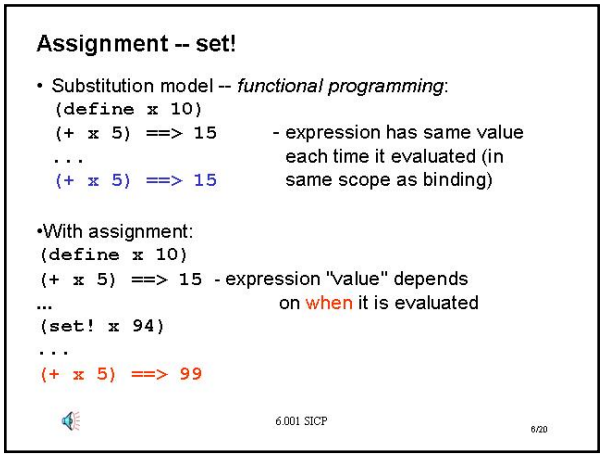


What does adding mutation do to our language? One of the primary effects of adding mutation is that we end up breaking our substitution model. This is okay, as we will shortly replace it with a better model.

In fact, the substitution model inherently assumed that we were dealing with **functional** programming, with no mutation or side effects. What does this mean? In essence, functional programming implies that we can conceptually treat our procedures as if they were mathematical functions. Yes, we know that we have procedures that deal with things other than numbers, but the same idea holds. This means that we could treat our procedures as **mappings** from input values to output values, and more importantly, that mapping was consistent, providing the same output value for a particular input value, no matter when we did the actual evaluation.

For example, if I define `x` to have the value 10, and then I evaluate the expression `(+ x 5)`, I of course get the value 15. If I then do some other computation and return to the evaluation of `(+ x 5)` I still get the value 15.

This means that this expression always has the same value, no matter when I evaluate it, and the procedure acts like a mapping from an input value to an output value, independent of time.



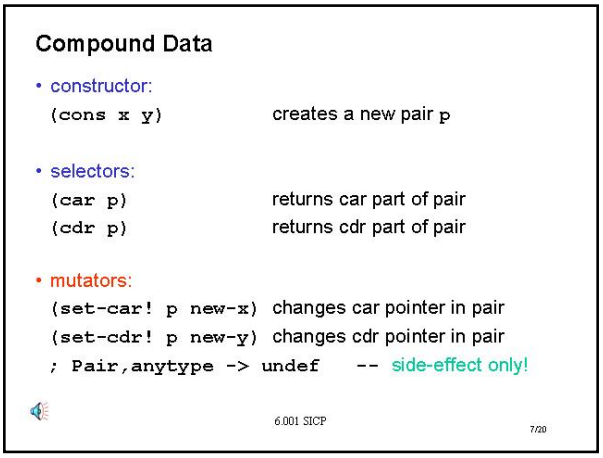
Slide 11.1.6

Once we introduce **mutation** or **assignment** into our language, this model no longer holds. In particular, an expressions value now depends on **when** it is evaluated, that is, what other expressions have been evaluated before it. In fact, notice the use of the term "when". We have now introduced **time** into our language in a very fundamental way. Now, two expressions with identical syntax may have different semantics, because they inherently rely on the context surrounding their evaluation.

To see this, consider the example shown. We again define `x` to have the value 10. If we immediately evaluate the expression `(+ x 5)` we will still get the value 15. Now suppose that at some future point, we mutate the value of `x` to have some new value, 94. Remember that `set!` finds the existing binding for `x` and changes it. If we then evaluate `(+ x 5)` we now get `99` as a value, since the value of `x` has changed.

Notice, we now have two syntactically identical expressions, but with different values or meanings or semantics. Thus, time now matters. Adding the ability to mutate a value means that context will influence the values of expressions. As we will see, having mutation makes some things much easier to do, but at the cost of greater potential for unanticipated effects.

Slide 11.1.7

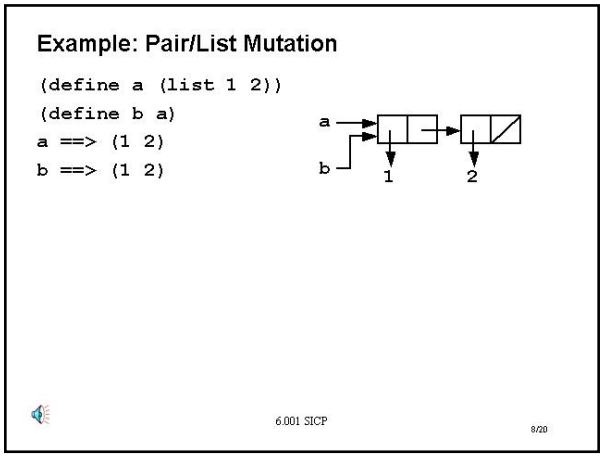


Not only can we mutate names, we can also mutate other basic data structures. For pairs, we also have procedures that change their pieces.

To remind you, the constructor for a pair is the primitive procedure `cons`, and the associated selectors are `car` and `cdr`. Now we introduce two mutators, one for each part of the data structure. Their forms are shown on the slide.

Both of these are normal procedures. Their behavior is to evaluate the first argument, which is expected to be a pair (or something made by `cons`). They also evaluate their second argument to get a new value or structure. The behavior is to then take the pair pointed to by the first argument and change or mutate either the car or cdr part of that pair (depending on which expression we are using) to point to the value of the second argument, thereby breaking the current pointer in the car or cdr part of the pair.

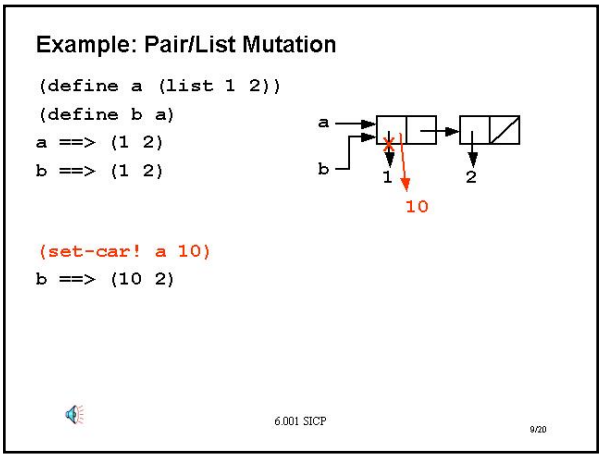
Notice the type definition of these procedures. In particular, the return type is unspecified, since the procedure is used strictly for the side effect of changing the pair structure.



Slide 11.1.8

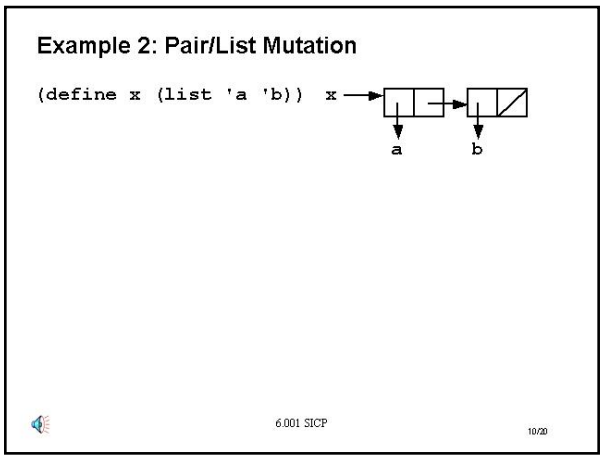
As we will see shortly, having mutation buys us some new power in our computational capabilities, but it also raises some new problems, which we want to highlight first. For example, suppose I create the list of 1 and 2, and give it the name `a`. I also give the name `b` to the same structure, and remember that the second `define` expression literally binds the name `b` to the value of `a`, which we know is the pointer to this box-and-pointer structure. Remember that there is no new creation of pairs in this case, `b` literally points to the same structure. If I ask for the value of `a` or `b`, in each case I get the list `(1 2)`.

Slide 11.1.9



Now suppose that some time later, I evaluate the expression shown in red. This gets the value of `a` which is the list structure in the upper right. It gets the value of the second argument, which is just 10. It then takes the `car` part of the box pointed to by `a`, breaks the current pointer in that box, and creates a new pointer to the new value, 10.

So what! Well, notice a subtle effect. If I ask for the value of `b` it has **changed!**. We get the value of `b` by tracing out the box-and-pointer structure, to get `(10 2)`. Yet nowhere in my code is there an explicit expression changing `b`. In this simple case, we know there is a tie between `a` and `b`, but in more general cases you can see how trouble can arise. Given the ability to share data structures, and to mutate data structures, one piece of code could change a value affecting some other variable without realizing it.

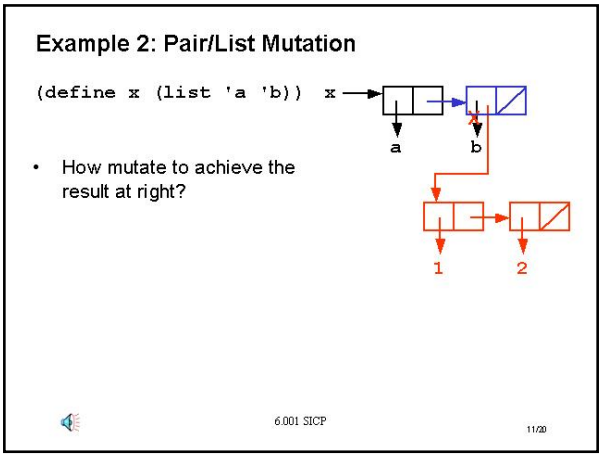


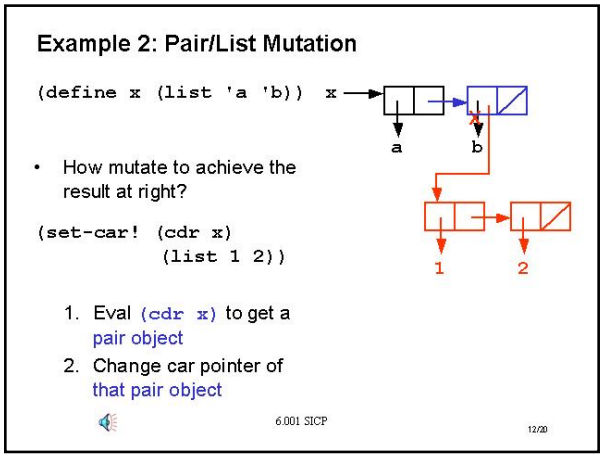
Slide 11.1.10

To use these mutators it is important to understand their affect on variables and structures, so that we can work backwards from a desired effect to determine the right mutation to cause that to happen. For example, consider the simple list structure shown here.

Slide 11.1.11

Now let's suppose I want to change the structure to have the form shown in red. What expression do I need to evaluate to cause this change? When you are ready to answer, go to the next slide.



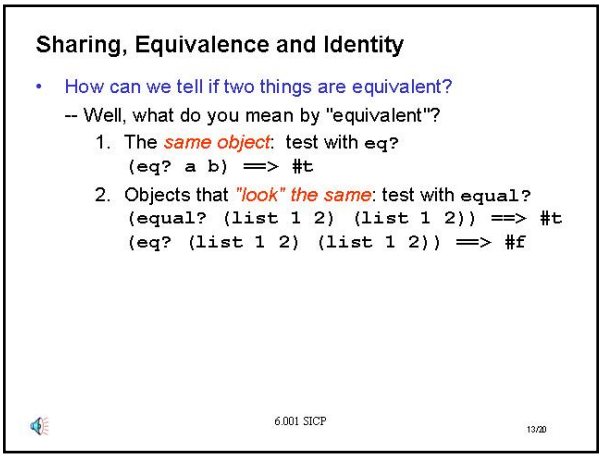


Slide 11.1.12

Here is the answer, and let's reason through why. First, we know that we want to change the `car` part of the second pair in the list. So, we need to evaluate `(cdr x)` to get to the second pair, the one shown in blue. Since we want to change the `car` part of this pair, we need to use `set-car!`. And what should the new value be? Just the list `(1 2)`, which we know we can create directly.

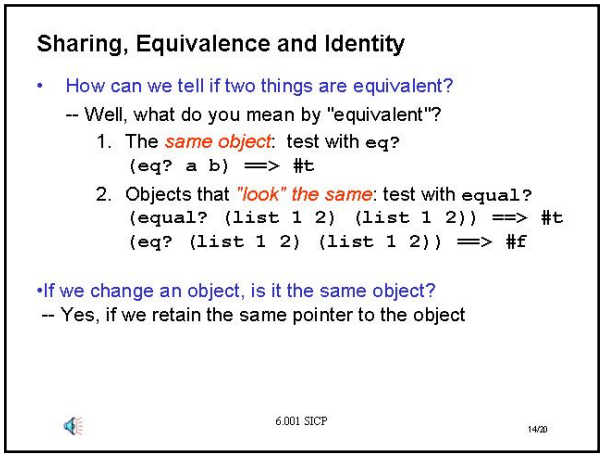
Slide 11.1.13

So it might occur to you to ask: "Given that different parts of a list structure can be changed by using mutation, how do we tell if two things are equivalent?" We already saw that we could have two different names for the same structure, and thus changing one name's value could cause the other name's value to also change.



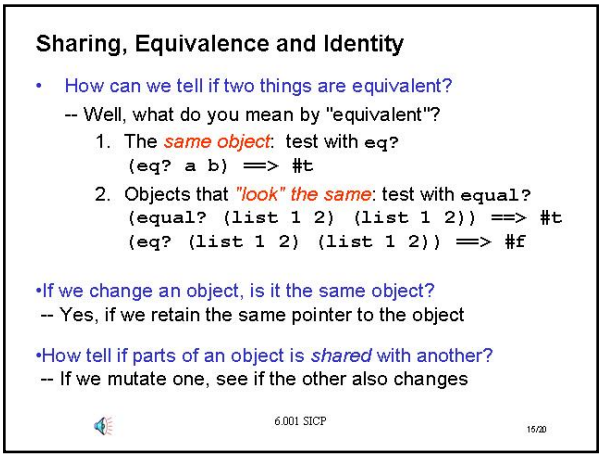
In fact, mutation actually causes us to first consider what **equivalent** means. If we want to know if two names point to **exactly** the same object, our test is `eq?`. This returns true, if the values of the two arguments point to **exactly** the same structure. Another way of saying that is that if we make some change to one structure, the corresponding change will be visible in the other structure. Thus `eq?` provides us with the finest level of detail in testing equality.

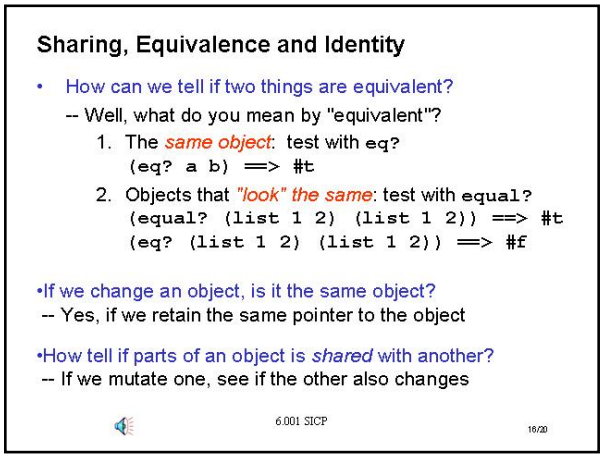
On the other hand, if we just want to know if two objects "look the same", that is, they print out the same form, then we use `equal?`. Thus, the test using `equal?` return true because these two `list` expressions result in structures that look the same. But we know that each call to `list` causes a new pair to be create so in fact these two `list` expressions create different box-and-pointer structures, and are not `eq?`. Thus, we have two different levels of granularity in deciding equivalence of structures.



Slide 11.1.14
These ideas of mutation and testing of equality raise some interesting issues. For example, if we mutate an object, do we still have the same object when we are done? The answer is **yes**, provided we maintain the same pointer to the object. For example, if we have some list structure with a pointer to its beginning, and we then mutate the contents of some of the cells in the structure, so long as we keep hold of the pointer to the beginning of the structure, we consider it to be the same. Its value has changed, but the object's identity is maintained.

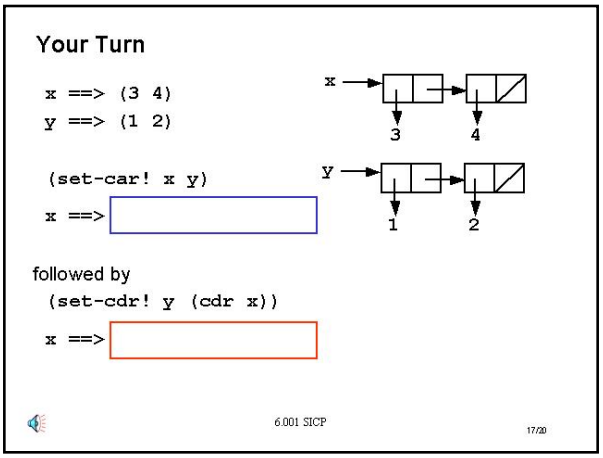
Slide 11.1.15
This tells us how to keep track of a particular object. A related question is deciding when two objects share a common part. The answer is that if we mutate one object, and see the same change occur in the other object, then we can deduce that these objects share parts. Because of our notion of equality as pointing to the same object, this means that changing that shared object through one name will be reflected when seen through the second name.

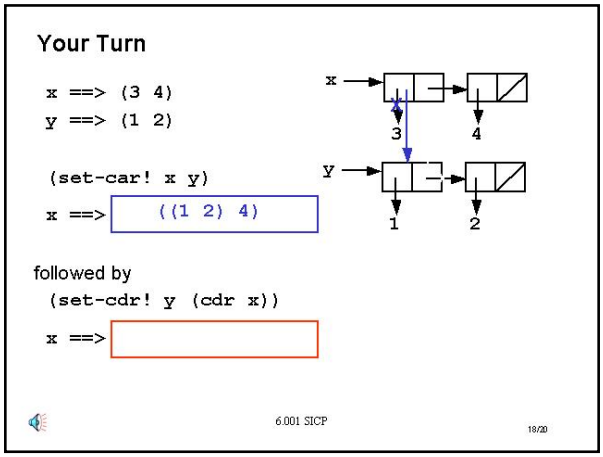




Slide 11.1.16
So what we see is that introducing mutation into our language has caused us to change how we think about equality, how we think about the finest level of detail in our representations. That has raised interesting issues about identity, equivalence, equality and sharing of structure.

Slide 11.1.17
To see if you are catching on to this, here are two definitions for list structure, with the names x and y. What is the value of x after each sequence of evaluations? When you are ready, move on to the next slide to check your answers.



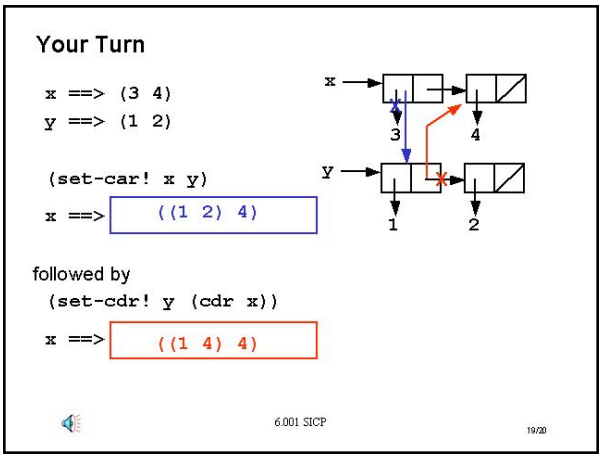


Slide 11.1.18

When we evaluate the `set-car!` expression, we first get the value of `x`, which we know points to the top list structure. We also evaluate `y`, which points to the second list structure. Then, we find the car box of the structure pointed to by `x`, and break the existing pointer in that box. We replace it with a pointer to the value of `y`, namely the second list structure as shown.

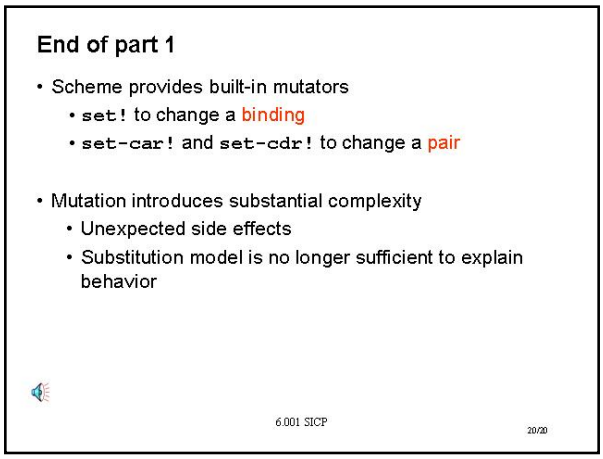
If we now evaluate `x`, we can see it points to a list of two elements. The first element happens to be a list of two elements, 1 and 2, and the second element is just the number 4. Thus, the value of `x` prints out as shown.

Slide 11.1.19



Now remember that time becomes important once mutation is allowed. Thus the change we have just made to `x` stays in place when we go to evaluate the next mutation. This says to get the value of `y`, which still points to the second list structure, and we break the pointer in the `cdr` part of the cons pair pointed to by `y`. In its place, we insert a pointer to the value of the second argument, which is just the `cdr` pointer of the cons pair pointed to by `x`.

If we then evaluate `x` again, we now get the form shown, simply by tracing through the list structure. Notice that due to the sharing of structure, the value of `x` has changed, even though no explicit expression involving a change in `x` was evaluated.



Slide 11.1.20

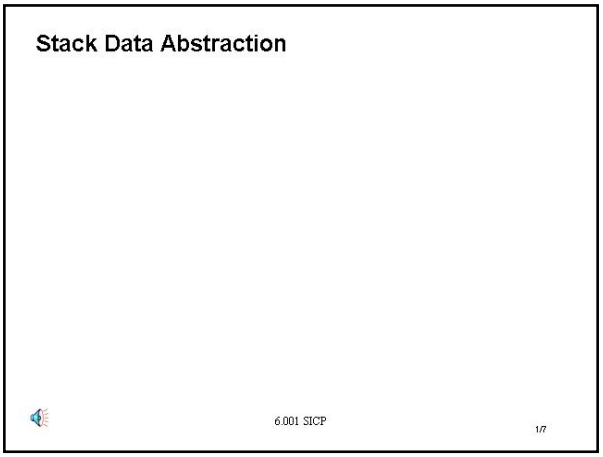
So here is a summary of the key points of this part of the lecture.

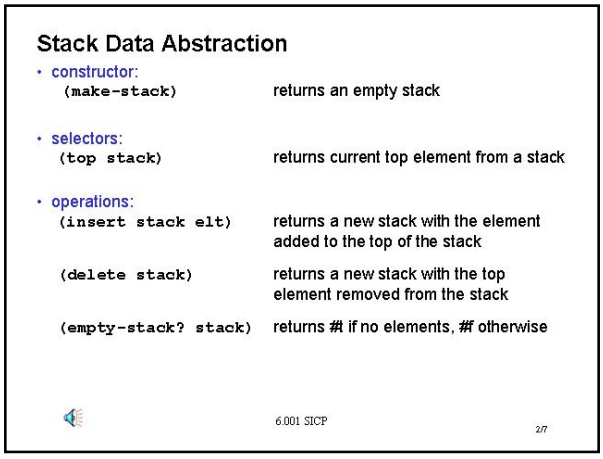
6.001 Notes: Section 11.2

Slide 11.2.1

Now, lets see how mutation can be used to build efficient data structures. We are first going to build a very useful data abstraction without mutation, then see how mutation changes its behavior.

The abstraction we are going to build is a **stack**, and it behaves just like a stack of dishes in a cafeteria. You can push a new plate onto the top of the stack and you can remove a plate from the top of the stack, but these are the only two operations you can execute on a stack. This is also referred to as a **last in, first out** data structure, for the obvious reason.





Slide 11.2.2

Here is our data abstraction template for a **stack**. Our constructor creates an empty stack, and our single selector gets the value of the top element of the stack.

The operations on the stack will help change its status. We have an operation for pushing a new element onto the top of the stack, returning a new stack. We have an operation for popping the top element from the stack, and returning the new, smaller stack. And we have a predicate for testing whether the stack is empty.

Notice that the selector gets the value of an element of the stack, while the operations create new stacks as a whole.

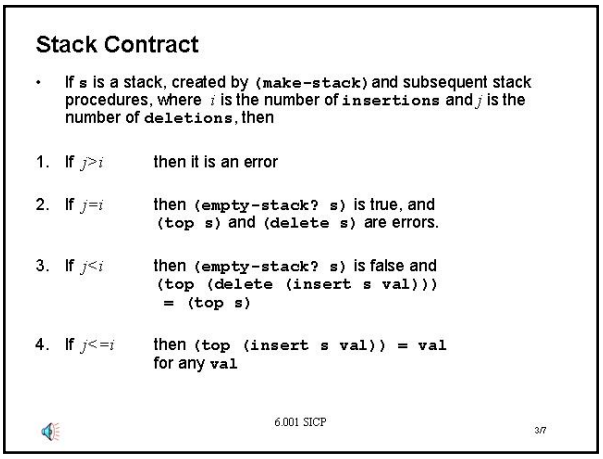
Slide 11.2.3

To be careful, we should define the contract for a stack, especially the relationship between the constructor, the selector and the operations that manipulate a stack. Here is a somewhat formal definition, but we can trace through the intuitive idea.

If we let **s** be a stack constructed initially to be empty, and we have performed **i** insertions, and **j** deletions, then here is the contract on the stack's behavior.

If we have tried to do more deletions than insertions, this must be an error; we can't more things out that we put in.

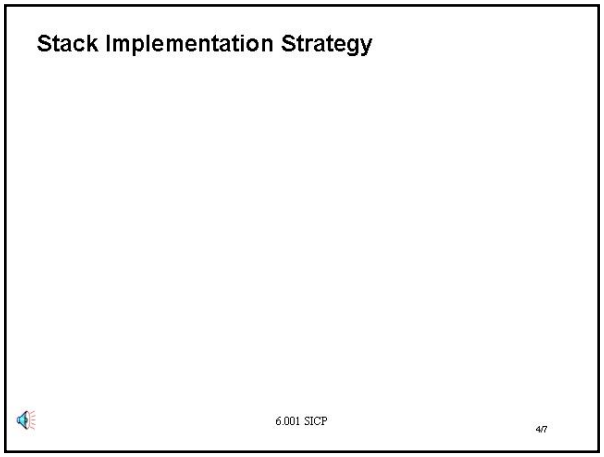
If we have made exactly the same number of insertions and deletions, then clearly the stack should be empty, and trying to either get the top element or delete something from the stack should be an error.



If on the other hand we have made more insertions than deletions, then clearly the stack should not be empty. More importantly, if we insert something, then delete it, the top element of the stack after this pair of operations should be the same as it was before this pair of operations. That is, nothing has changed in the stack below this point.

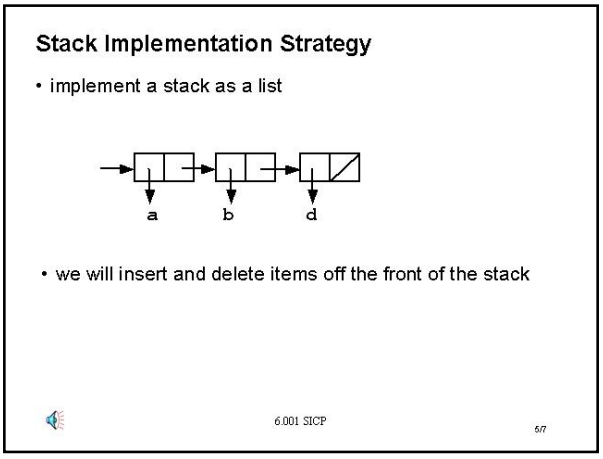
Finally, if we have a legal stack, then if we push an element onto the stack and look at the top element of the stack, we see exactly this element that we just pushed.

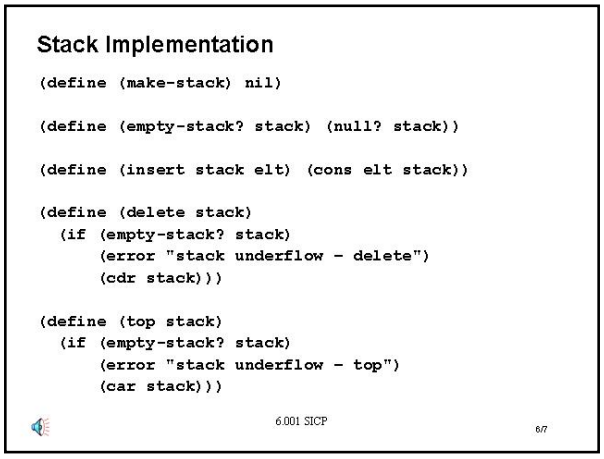
So we see that the stack contract describes exactly the "last in, first out" behave we want.



Slide 11.2.4
Having defined a set of operations on our abstraction, and a contract on the behavior of the abstraction and its associated operations, we can actually implement this ADT. Our first strategy will be to treat a stack as a list.

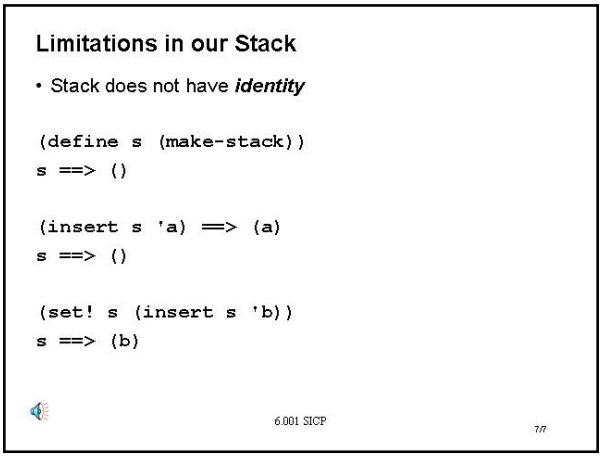
Slide 11.2.5
For example, this list could represent a stack, where **a** is the most recent thing pushed onto the stack, with earlier pushed elements consecutively arrayed down the list.
Then to insert and delete elements from the stack, we simply add things to the front of the list, or take the cdr of the list.





Slide 11.2.6
So here is an implementation that uses lists to incorporate this idea. Note how the constructor and predicate simply build on the underlying use of a list.
Insertion simply conses a new element onto the existing stack (or list), returning a new list with that element at the top of the stack (or front of the list).
Notice the defensive programming used for deletion or checking the top, in which we ensure we have a legal structure before attempting to access the list structure. If we do, then we either use `car` to get a pointer to the first element, or we use `cdr` to get a pointer to everything **but** the first element.
Check to convince yourself that the contract holds for stacks, by building on the contract for list operations.

Slide 11.2.7



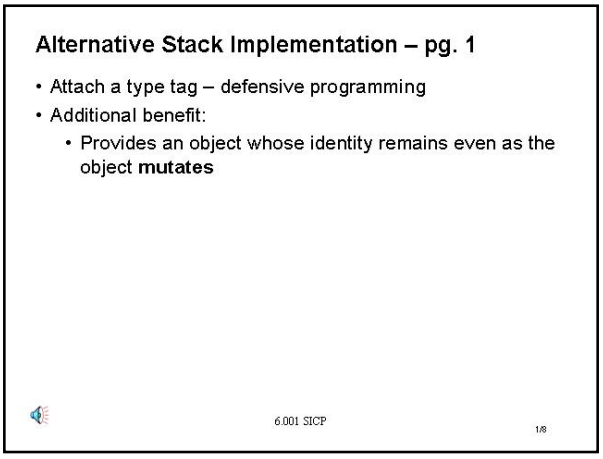
While this looks fine, this implementation strategy has a problem: our stacks do not have an identity. What does this mean?

Well consider the example shown. I can first create a stack, and give it a name. Now I insert the element **a** into that stack, and you can see it returns for me the stack with only the element **a** in it. But if I ask for the value of **s** I get back the empty list, not this stack? Unfortunately while I created a new list when I did the insertion, I didn't actually change the value pointed to by **s**, and thus that remains the empty stack.

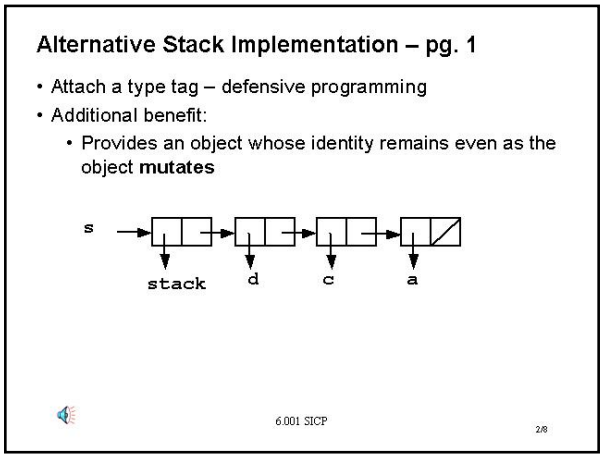
Worse yet, I can manipulate pieces of the stack directly. If I insert **b** into **s**, I should get a stack with **b** and **a** in it. But I can mutate the name of the stack to point to only part of this stack, thus violated my contract. **The problem is that I have not isolated the stack from outside use. I would really like the stack to keep its identity, and to be manipulated only by the contract operations, and we will turn to that in the next section.**

6.001 Notes: Section 11.3

Slide 11.3.1



The first thing we need to do is practice defensive programming, that is, put a tag at the front of the structure to identify it. One additional advantage is that the identity of this structure will remain the same, even if pieces of it mutate.

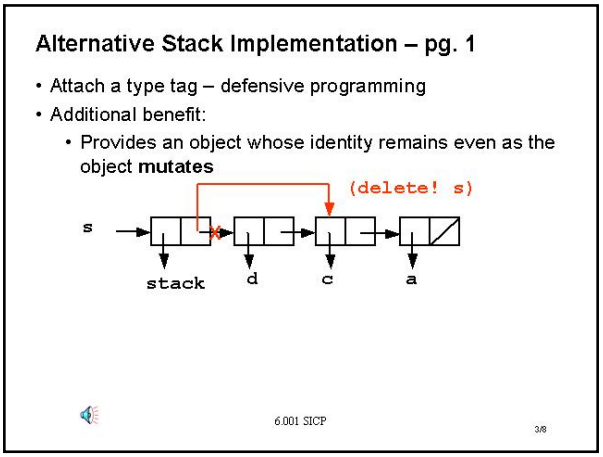


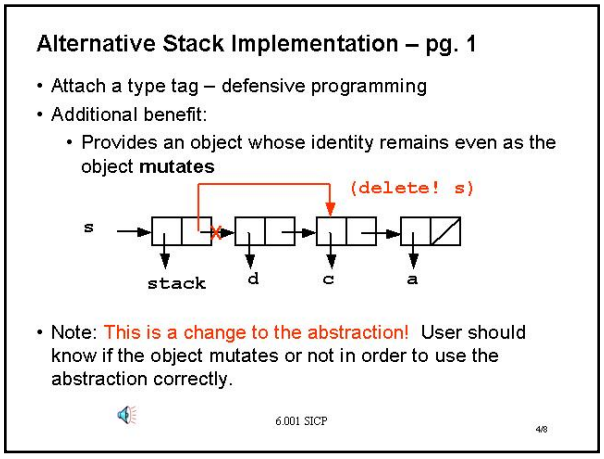
Slide 11.3.2

What does that mean? Well, putting a tag on the front in the standard way would result in our object consisting of a list whose first element is the symbolic tag, and whose `cdr` points to the actual contents of the stack.

Slide 11.3.3

Now, if I decide to delete an element from this stack, my operation should get to the actual stack (everything but the tag), mutate that structure to drop the first element, and reattach that stack to the tag. Notice that by doing this, **s** still points to the same structure, that is to the cell with the symbolic tag and a pointer to the stack contents. This fixes the problem I saw previously, by maintaining a notion of the identity of the stack.





Slide 11.3.4

Notice that this is really a change in the contract. We should really advise the user that the stack is mutating, as this is exactly what we have proposed here. We are proposing to mutate the stack to preserve the pointer from the stack's name to the tagged list, while mutating the part of the list that represents the stack's contents.

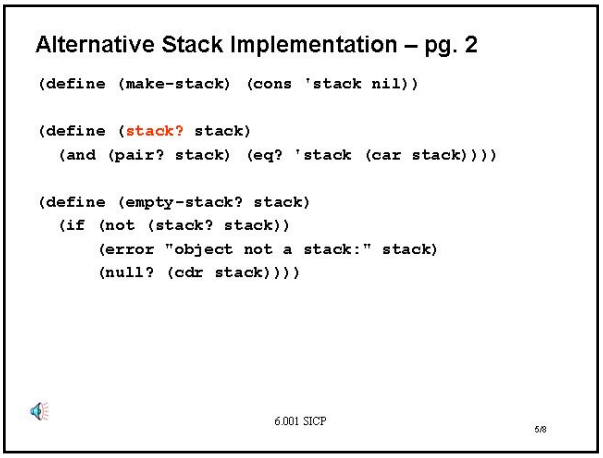
Slide 11.3.5

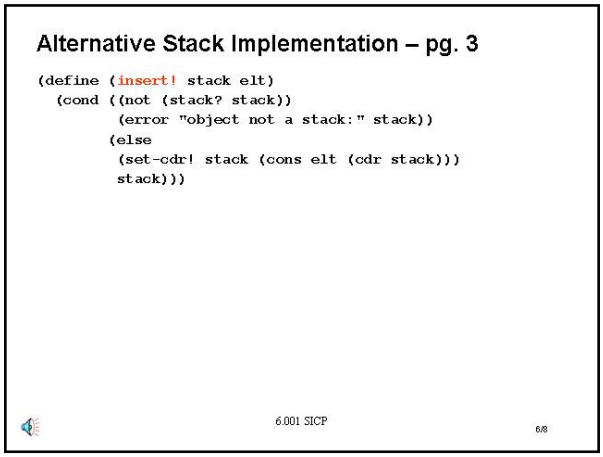
So let's change our implementation to capture this idea.

First, our constructor must now create a tagged representation, where the actual contents are still an empty list. Note that this is gluing together two different things, a symbolic tag, and an actual stack.

Given that, we can now create a predicate for stacks. Notice the defensive programming in which we first ensure that the argument is a pair or list, before checking to see if the first element is the type tag.

We can also build our predicate for the empty stack. Notice that it first checks to see that we have a tagged data structure of the appropriate kind. If we do, then we don't have to further check to see if the object is a list that happened as part of the type check. So we can immediately proceed to the `cdr` of the argument, and check its contents.





Slide 11.3.6

Our operations on stacks will now be mutators, which I will denote by appending a `!` to the end of the procedure's name.

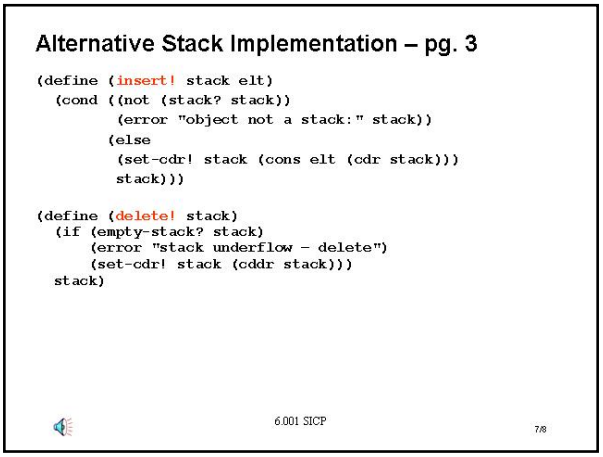
`Insert!` first does some defensive programming; to make sure that the argument passed in is labeled as a stack.

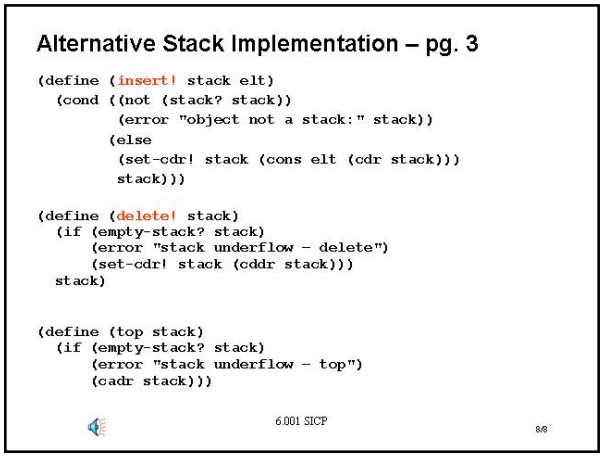
If we have a stack, then notice what we do. We get the `cdr` of the stack, which is the pointer to the list representing the actual contents of the stack. We cons a new element onto that list, putting that element at the front of the stack. `Cons` returns a pointer to the beginning of this new list, and we now mutate the `cdr` pointer of the stack itself to point to this list. Thus, we have changed the contents of the stack. Finally, we return the value of the argument, since that gives us the labeled data structure.

Notice how identity is preserved in this manner. The argument passed in points to some list structure, and when done, we return the same pointer. The only issue is that some of the contents within that list have changed.

Slide 11.3.7

Delete has a very similar form, which you can check through. Use a box-and-pointer diagram to check your reasoning.





Slide 11.3.8

To get the top element of the stack, we just check that we have a legal stack, and then get the value at the top of the contents of the stack.

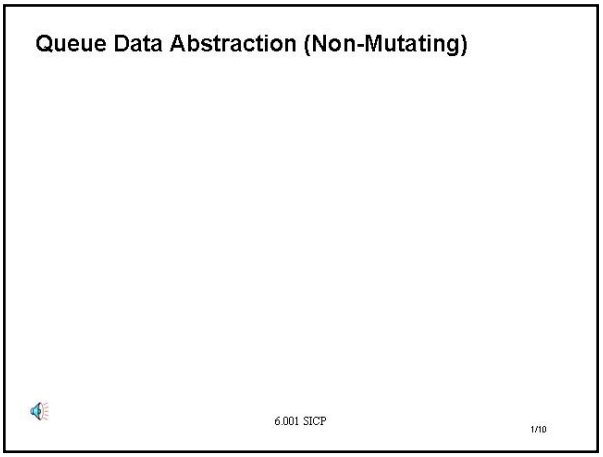
Notice how mutation allows us to create structures that are efficiently represented, yet allow us to maintain the identity of data objects.

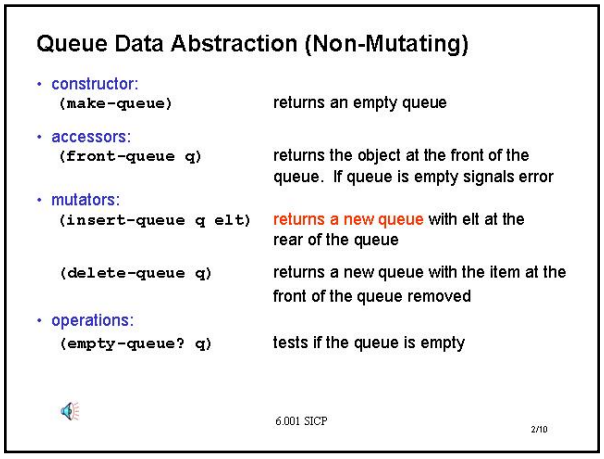
6.001 Notes: Section 11.4

Slide 11.4.1

We have seen how mutation can give us different ways to build data abstractions. Let's push further on this idea by looking at a new data abstraction, called a **queue**.

A queue behaves like a line, say the line in front of a movie theatre. Thus, one adds things to the end of the line, but takes things from the front of the line. This gives a different behavior than a stack. Remember that a stack was a "last in, first out" structure, whereas a queue is a "first in, first out" data structure. In other words, the first element placed in a queue will be the first element taken out of the queue.





Slide 11.4.2

Here is the template for our data abstraction, with a constructor, a selector for getting the front element of the queue. We will have two operations that change the queue (though in this version, they won't use built in mutation to do this). The first will insert an element at the end of the queue, but will do this by actually making a new queue. The second will return a new queue with the first element removed. And we will have a way of testing whether the queue is empty.

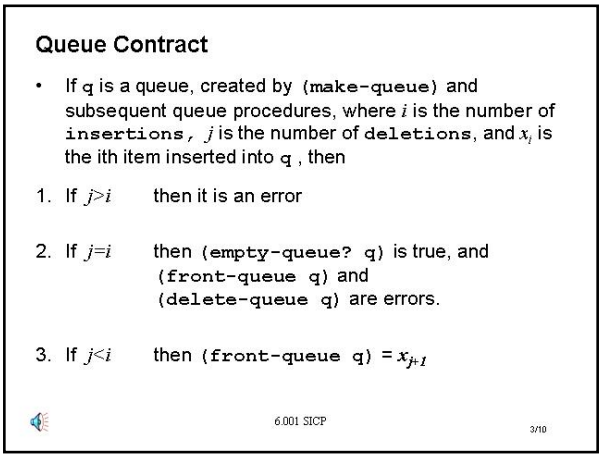
Slide 11.4.3

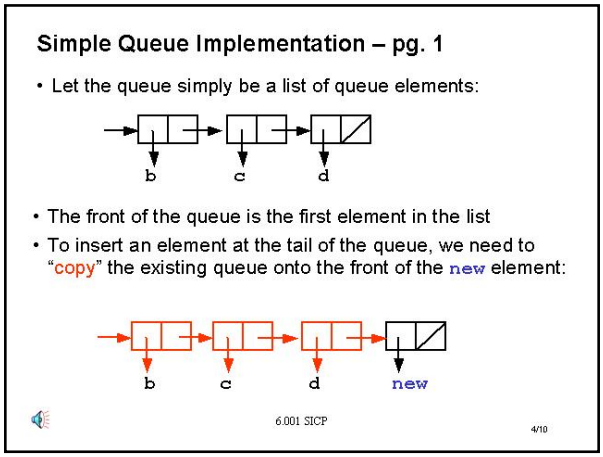
As with stacks, we will have a contract that governs the behavior of the abstraction. With the notation shown, we can see that the behavior can be defined as follows.

If we try to take more things out of the queue than we have put in, we will get an error.

If we have taken out exactly as many things as we have put in, then the queue is empty, and trying to either remove something or look at the front of the queue will be an error.

Finally, if we have put more things into the queue than we have removed, and we have removed **j** things, then the element at the front of the queue will be the **j+1**st thing inserted, i.e., the next one in order.





Slide 11.4.4

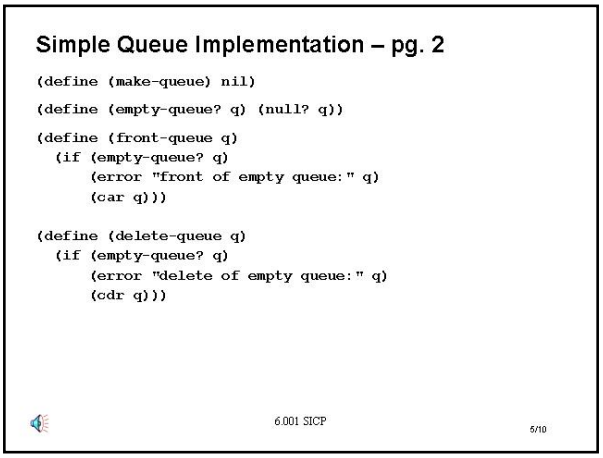
Just as we did with stacks, let's start with a simple implementation of a queue. In particular, we can just represent a queue as a list of elements. Here we can decide that the first element of the list also represents the first element of the queue. In that case, getting the front of the queue is easy; it is just the `car` of the list. Similarly, deleting an element from the queue would be easy, we would just take the `cdr` of the list to remove the first element of the queue.

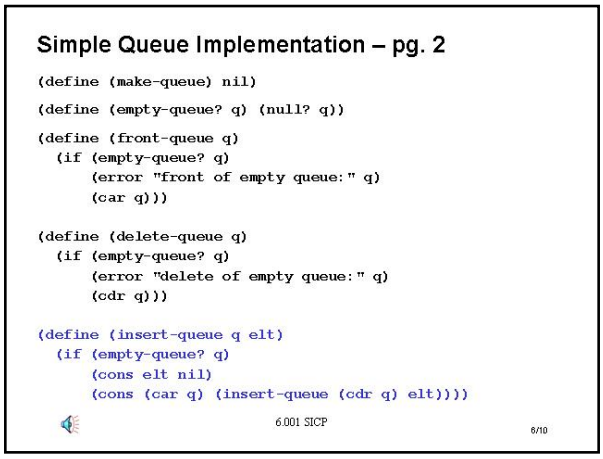
But notice that to add an element to the queue is going to take some effort. To do this, we need to place that new element at the end of the list, and to do that; we need to first **copy** the entire list until we reach the end, then "cons" onto the last element of the list a pointer to a list containing the new element. This will give us the correct structure, a list in order, with the new element as the last thing in the list.

So how do we implement this?

Slide 11.4.5

Since we are using a list as our internal representation, our constructor just makes an empty list as an empty queue, and testing for an empty queue uses the associated list predicate. Finding the front of the queue or deleting the front element from the queue have a similar form. We first check that we have a non-empty queue. If we do, we can use the associated list operation. We use `car` to get the first element, which we know is the front of the queue, or we use `cdr` to get the rest of the queue, except the first element.



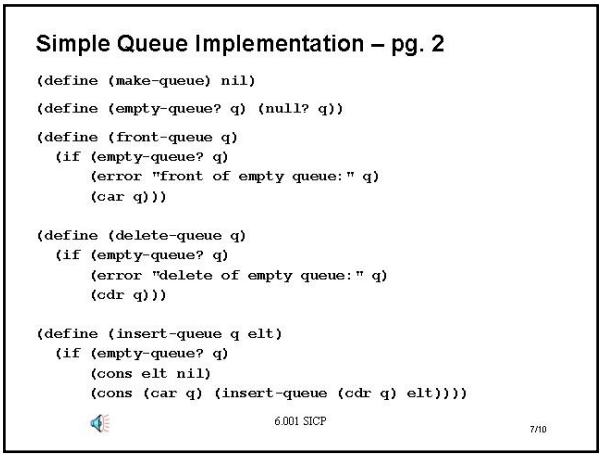


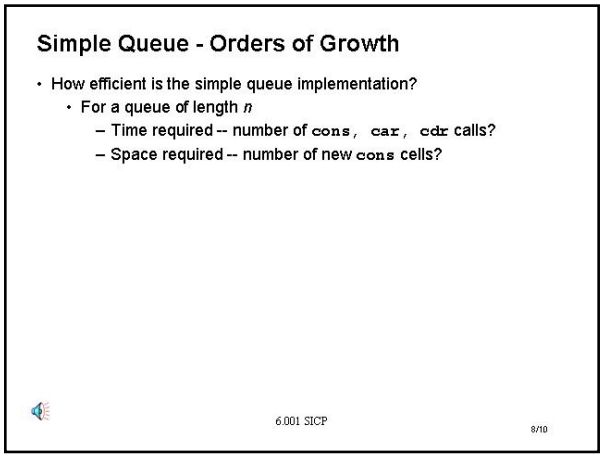
Slide 11.4.6

Inserting an element into the queue is a bit more of a pain in this implementation. Remember that this element has to go at the end of the current list representing the queue. Thus our procedure recursively walks down the list representing the queue, until it reaches the end, at which stage it creates a list representing a queue with this element. Notice how we make a copy of the entire queue so that the result is a **new** list, with that element now at the end.

Slide 11.4.7

You can now see that while this implementation satisfies our contract for a queue, it comes with a cost in terms of efficiency.



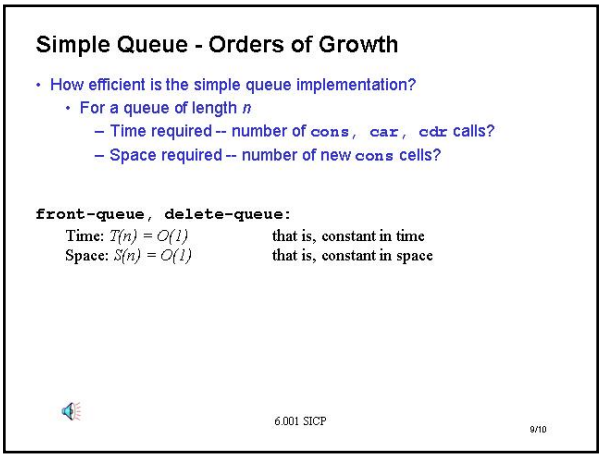


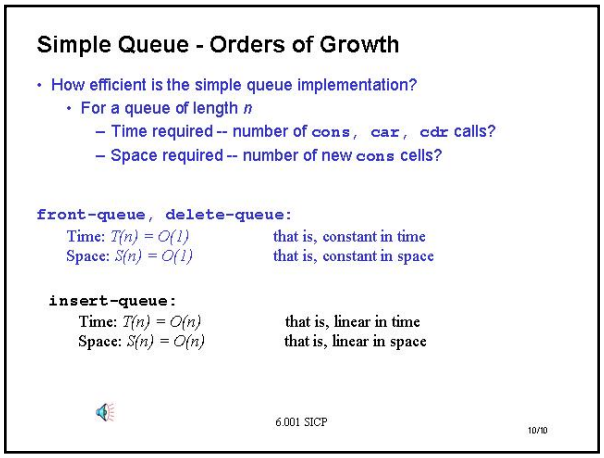
Slide 11.4.8

Let's actually measure that cost. If we have a queue of length **n**, let's measure the efficiency in both time and space for manipulating such a queue. In terms of time, we want to measure the number of primitive list operations, that is, `cons`, `car` and `cdr` evaluations that are needed, and in terms of space, we want to measure how many new pairs or cons cells are created when we use one of our queue operations.

Slide 11.4.9

Let's start with the easy ones. To find the front element of the queue takes time that is constant in the size of the queue, since we just use `car` to get at it. Similarly, to delete an element from the queue, we just use `cdr`, which is also constant in time. Both operations are also constant in space, since no new cons cells are generated.



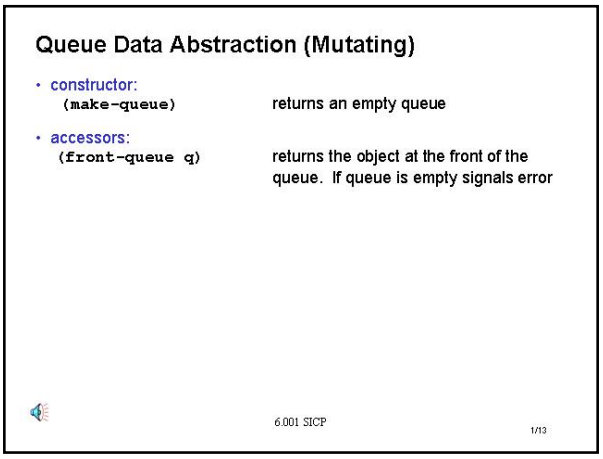


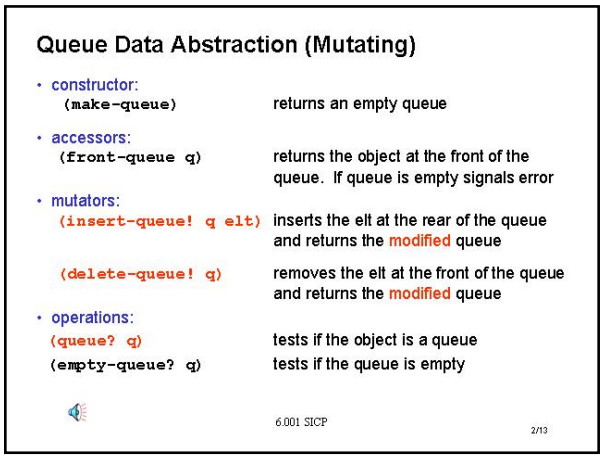
Slide 11.4.10

As we have already suggested, though, insertion into a queue is more expensive. As we have seen, to add a new element, we have to walk our way down the list, one element at a time, until we reach the end of the list. Thus, this operation will take time linear in the size of the list. And since we must also make a copy of that list as we do this, we cons up a number of new pairs that is also linear in the size of the list.
The question is whether we can do better than this?

6.001 Notes: Section 11.5

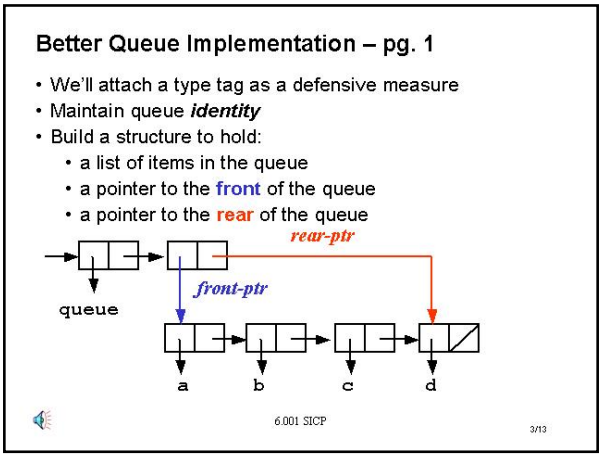
Slide 11.5.1
Let's go back and look at queues again, now seeing how mutation can allow us to build a more efficient implementation of this data structure.
As before, our constructor will build an empty queue and our accessor will get out the front element of the queue.

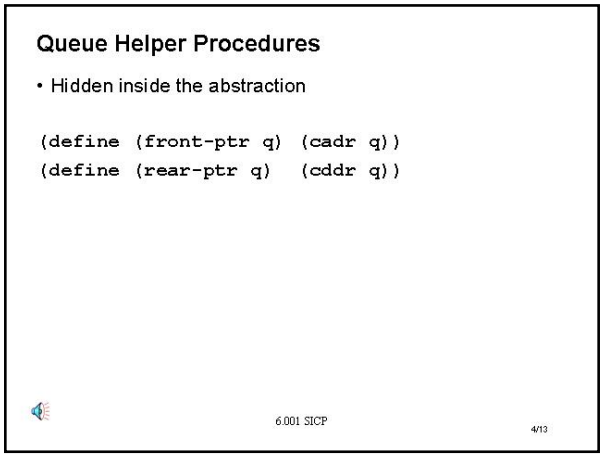




Slide 11.5.2
The big change we are going to make deals with how we add things or remove things from the queue. Rather than making a copy of the queue, as we did before, in order to add something to the end, instead we are going to directly modify the list structure representing the queue, changing the pointer at the end of the old queue to now point to the most recent insertion. Thus, we will mutate the existing structure rather than copying it. Similarly, we will use mutation for deletion of elements in the queue, so that the internal list representation will be directly modified in this case as well.

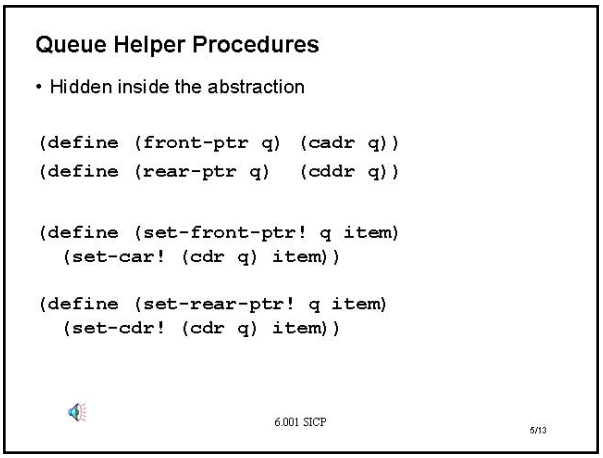
Slide 11.5.3
Here is our strategy. As in the previous case, we will attach a type tag to the front of the structure, as a defensive measure. This will let us identify when an object is a legal queue. This will also provides us with a method for maintaining the **identity** of a queue as we mutate its contents.
The second thing we will do is build a structure that incorporates a bit more information than the last version. This structure will contain a pointer to the list that represents the contents of the queue, but it will also include a pointer to the **front** of the queue and a pointer to the **rear** of the queue.
We have lots of choices for how to capture this information, but the easiest one is that shown in the diagram. Thus, a queue data abstraction points to a list, whose first element is the type tag. The second part of that pair points to a new pair, which contains two important pointers. The `car` points to the pair at the beginning of the list representing the contents of the queue. The `cdr` points to the pair at the end of that list. Notice that the list is still used to represent the queue's contents, but now we have direct access to the first and last element in the queue, and this will allow us to build much more efficient implementations of queue operations.

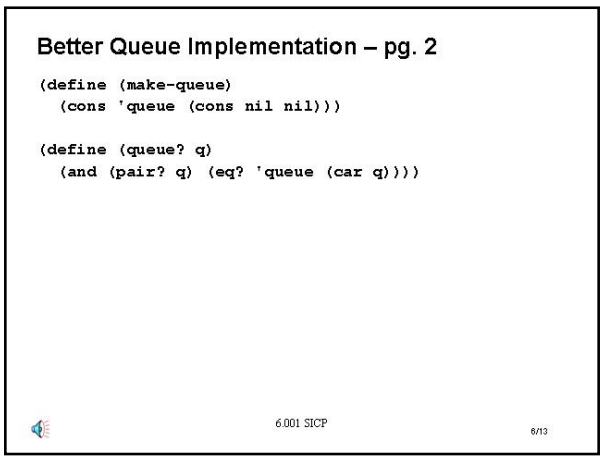




Slide 11.5.4
Given that idea, we will need some procedures to manipulate the parts of a queue. These procedures should be hidden inside the abstraction. This means that they should only be used within procedures designed to manipulate the queue, and they should not be accessible to consumers of queues.
First, we will need procedures that get the pointer to the first and last pair in the list of contents. Given our design on the previous slide, we can see that these two procedures will strip off the type tag, (using `cdr`) then access the contents of the `car` or `cdr` part of the resulting pair. This will give us a pointer to the **cons cell** at the beginning or at the end of the list representing the queue's contents. Note, it does not give us the actual value; it gives us a pointer to the cons cell, for reasons that will be clear shortly.

Slide 11.5.5
Given that we have access to the front and rear cells in the list, we also want the ability to change them, so we have two mutation functions.
To change the front pointer of the queue, we will do the following. First, we take the `cdr` of the queue. Remember that this gives us the box containing the front and rear pointers. We then **mutate** the `car` of that box (the front pointer) to now point to a new item, which we assume is the beginning of a list of elements. A similar form holds for changing the rear pointer of the queue.

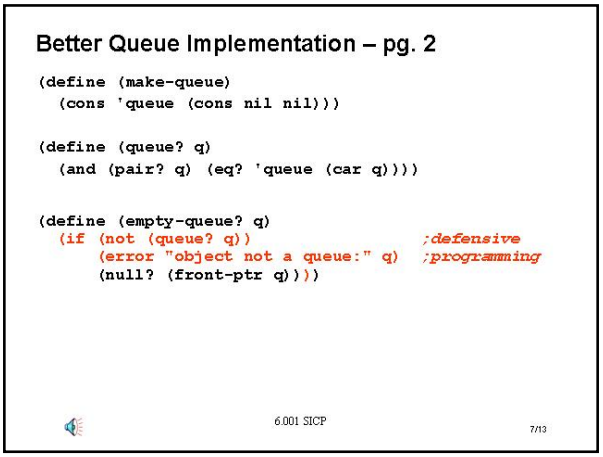


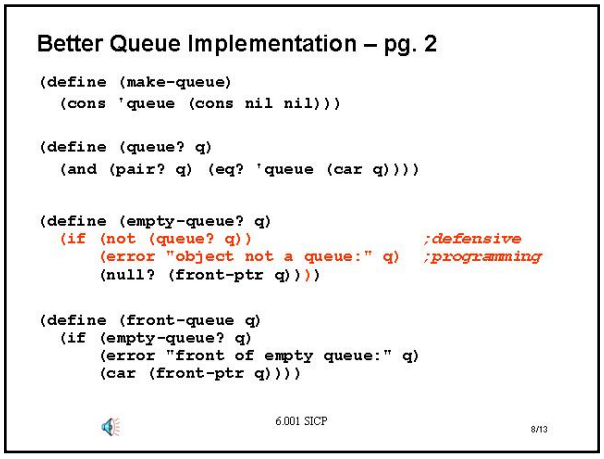


Slide 11.5.6
Using these ideas, we can build a much better queue implementation.
To create an empty queue, we need to set up this structure. Note that we first "cons" together two empty lists, to represent front and rear pointers to an empty queue. Then we attach the type tag to the front of this, and return this list structure as our queue representation. You might draw a box-and-pointer diagram to convince yourself that this generates the structure we illustrated in the previous slide.
With this defensive programming construct in place, we can utilize it when checking for a queue. We confirm that we have a pair, and then check whether the `car` is the tag for a queue.

Slide 11.5.7

To see if we have an empty queue, we can now be clever. We first check to be sure the argument is a queue, using our defensive programming style. Then, we use our internal procedure to extract the front pointer of the queue (the pointer to the beginning of the list of elements) and check to see if it is empty.





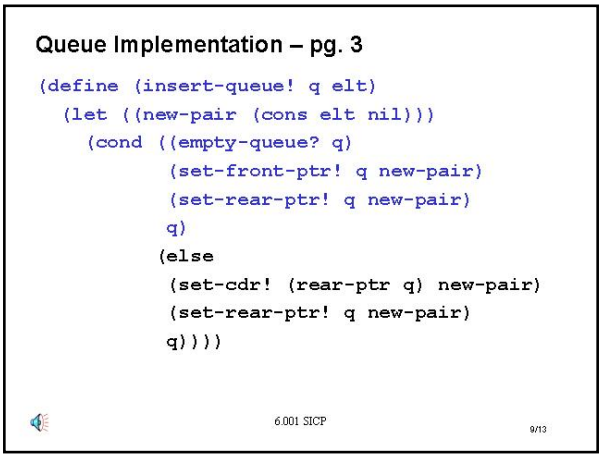
Slide 11.5.8

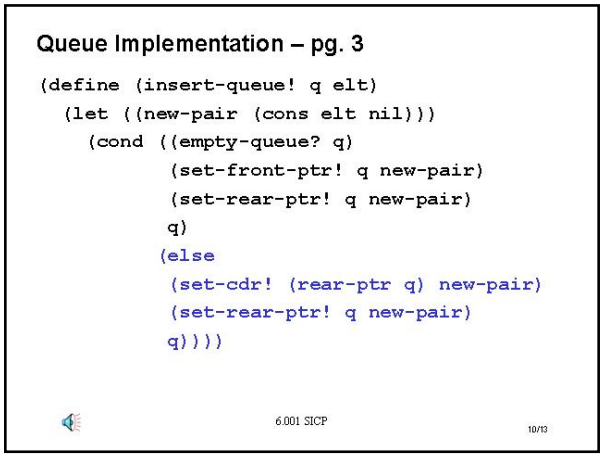
Finding the front of the queue (that is the first element) is almost the same. The only difference is that once we find the pair pointed to by the front pointer, we extract the `car` to get the actual element.

Slide 11.5.9

The tricky part arises when we want to actually manipulate the queue, when we want to modify the structure representing the contents of the queue. To insert a new element into the queue, we first create a list of just this object. Notice that this becomes the portion of the list that should be end of the entire list of contents.

If the queue is currently empty, then this new list is exactly what I want for my queue, so I simply change the front and rear pointers of the queue to point to this cell (since there is only one element, it is both the front and rear of the queue). Then I return the pointer to the actual queue.





Slide 11.5.10

If the queue is not empty, then I need to execute the following steps. First, I get the rear pointer of the queue (remember that this is a pointer to the last cons cell in the list representing the queue's contents). Then, I mutate the cdr of this cell (which is currently the empty list) to point to the new pair. Notice what this does, it results in a list one element longer, but without having to make a copy of the list or walk down the length of the list.

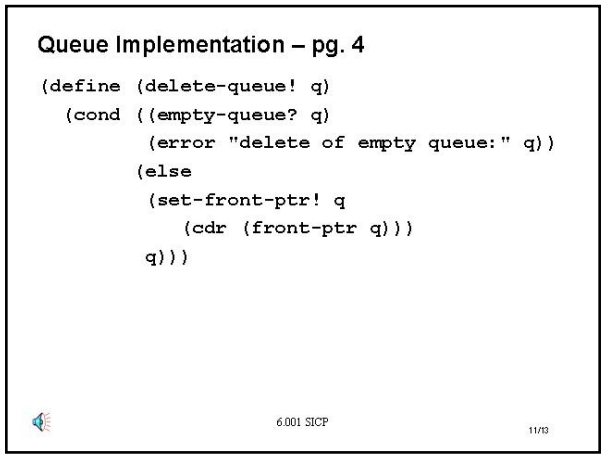
Finally, I need to preserve the correct information about the queue, and my rear pointer is now out of date. So I modify that pointer to now point to the last cell in the new list. And then I return the pointer to the whole queue structure.

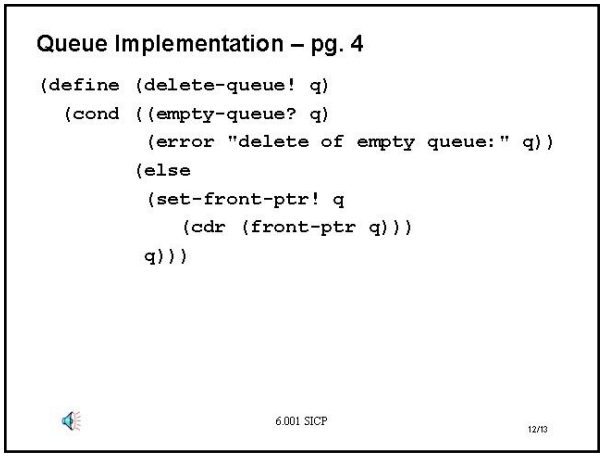
Slide 11.5.11

Deleting an element of the queue is very similar. If the queue is empty, we complain.

Otherwise, we get the front pointer of the queue (this is the first cons cell in the list) and we take its `cdr`, which thus points to the rest of the list, other than the first element. We then modify the front pointer of the overall structure to point to this new list, now removing the first element from view.

To convince yourself this is correct, try an example with a box-and-pointer diagram.





Slide 11.5.12

What are the orders of growth now? Well in this case the two previously expensive operations are now both constant in time and space. Thus, mutation has given us a more efficient way of representing the same data abstraction.

Slide 11.5.13

Here is a summary of the key points of this lecture.