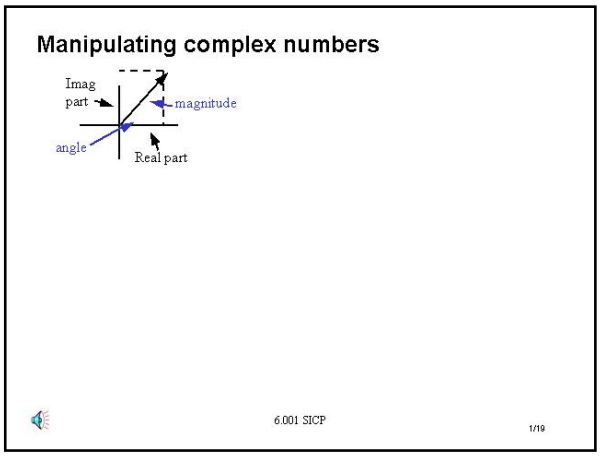
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

6.001 Notes: Section 9.1

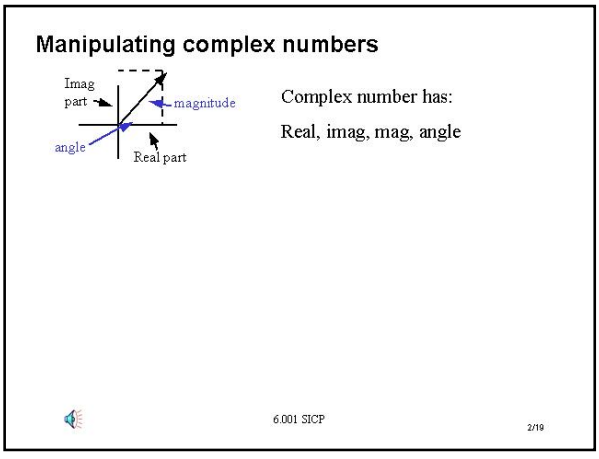
Slide 9.1.1



In the last lecture, we introduced **symbols** into our language. And we showed how to intermix them with numbers to create mixed expressions. We saw how to use that idea to create a symbolic differentiation program that could reason about algebraic expressions, rather than just numeric expressions. In this lecture, we are going to take the idea of symbols, and combine them with lots of different data structures, to create **tagged data structures**.

Why do we need a tag? .. and what is a tag? We'll answer these questions by considering an example. Suppose I want to build a system to manipulate complex numbers. Remember that a complex number is a number with two parts, standardly represented as a **real** and an **imaginary** part. Think of these as two axes in a vector space, or as a point in the plane (though we will see that there are special rules for how to manipulate such points that are different from normal vectors).

Because we can represent a complex number as a vector, we can also think about representing such numbers in terms of a **magnitude (or length)** and an **angle** (typically with respect to the real axis).



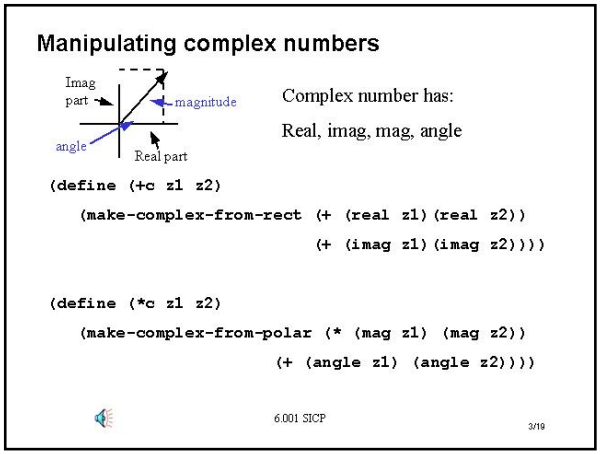
Slide 9.1.2

Now, let's assume we have some data abstractions for complex numbers. We have a constructor, and appropriate selectors, including selectors for both the real and imaginary part, and for the magnitude and angle.

As we saw earlier, given such constructors and selectors, we can proceed to write code to manipulate complex numbers, without worrying about internal details. So let's do that ...

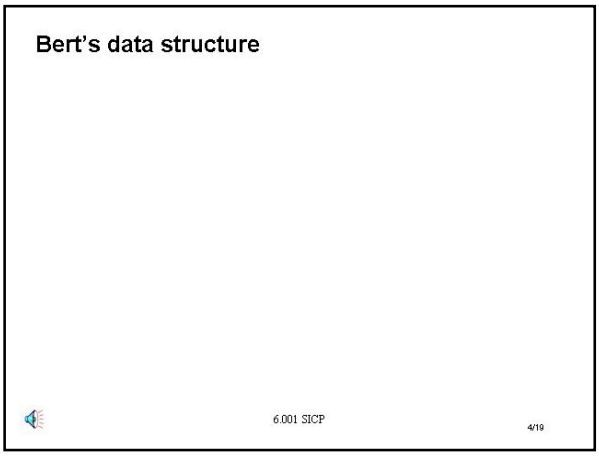
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 9.1.3



When manipulating complex numbers, I can take advantage of the fact that some things are easy to do in Cartesian (or real and imaginary) coordinates, and some things are easy to do in polar (or magnitude and angle) coordinates. For example, adding two complex numbers is most easily conceptualized in Cartesian coordinates, while multiplying two complex numbers is most easily conceptualized in polar coordinates. Addition is in fact just vector addition, so I use the selectors to get the parts of each complex number, add them together using standard addition, then glue the two parts together to make a new complex number. Here I will need to use a constructor that is making a complex number given Cartesian coordinates as input. For multiplication, I will use the polar selectors to get out the parts, so that I can just use normal multiplication on the magnitudes, and normal addition on the angles to get the new components. Here, I will use a constructor to make a new complex number that knows its inputs are provided in polar coordinates.

Note once more how we are separating the use of a data abstraction from its implementation. I can write code that uses complex numbers, while assuming that someone else will eventually provide a specific implementation. And note the standard form of such procedures: we use selectors to get out the parts, apply more primitive operations to those parts, then reglue the parts back into a data abstraction.

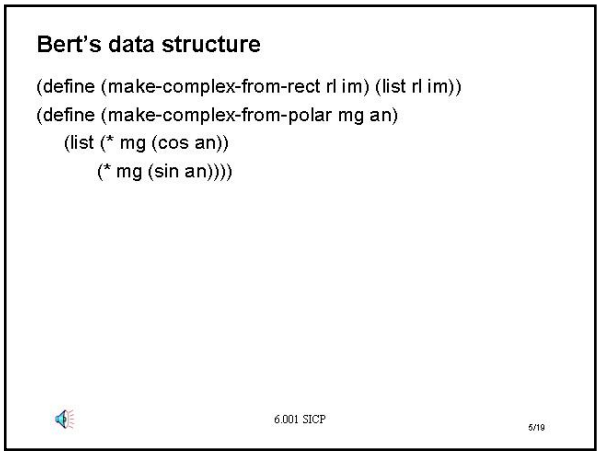


Slide 9.1.4

While it is convenient to separate data abstraction implementation from data abstraction use, ultimately we will have to provide a detailed implementation, that is, we will need to build the "guts" of this abstraction.

Suppose we ask our friend Bert to provide an implementation for complex numbers...

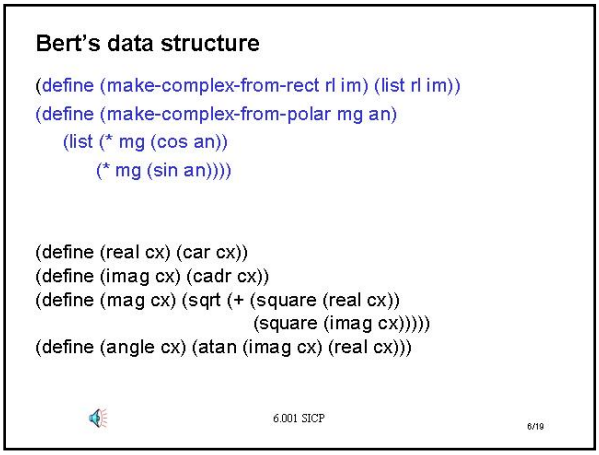
Slide 9.1.5



First, Bert will need a way of gluing things together. He decides, rather naturally, just to use lists to glue pieces together. He still has a choice, though, and being a rather "square " guy, Bert simply opts to use rectangular coordinates as his basis. That means that Bert's representation of a complex number is as a list of a real and imaginary part.

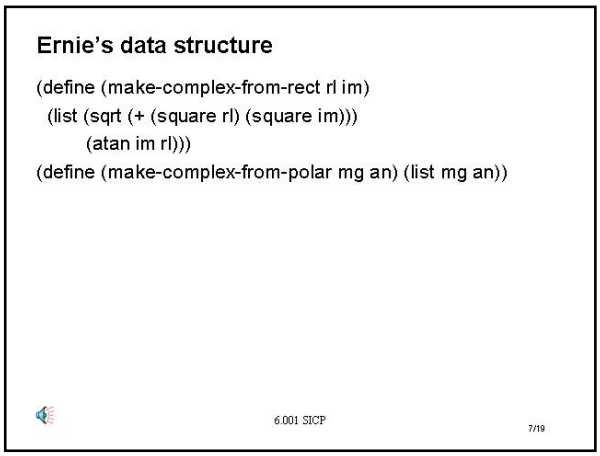
Note what this means. If we hand Bert a real and imaginary part for a new complex number, he can simply glue them together using `list` as this directly meets his representation. On the other hand, if we hand Bert a magnitude and angle, he will need to convert these to real and imaginary equivalents (using the appropriate trigonometry) so that he can then make a list of the real and imaginary part, which is how he represents these numbers. Thus, Bert's internal representation is always as a list of real and imaginary parts.

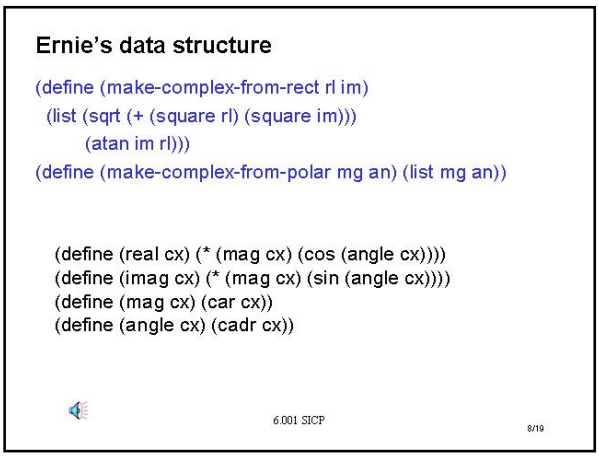
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.1.6
To complete the representation, Bert just has to ensure that he implements selectors that together with the constructors meet the contract for complex number abstractions. For `real` and `imag` parts, this is easy, as we can just rely on the contract for lists. For `mag` and `angle`, however, we must get out the real and imaginary parts (since these are the underlying representational pieces) using the appropriate selectors, then compute the appropriate values from those parts.
This then completes Bert's implementation.

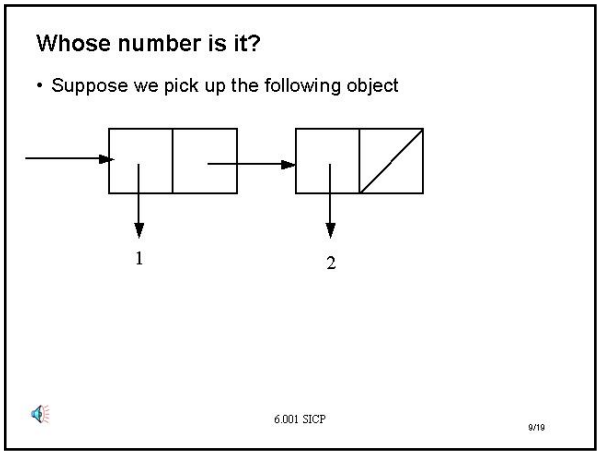
Slide 9.1.7
Notice that Bert made a design choice. He made a decision to represent complex numbers by a list of their real and imaginary parts. Let's suppose at the same time we also asked Bert's friend, Ernie, to create an implementation of complex numbers. Since Ernie is Canadian, he likes cold weather, and thus decides to implement complex numbers in polar form. Thus, his basic representation is a list of magnitude and angle, which means his constructor for polar form is just a list, but if he is given a real and imaginary part, he will first have to convert them to polar form, then store those values as a list.



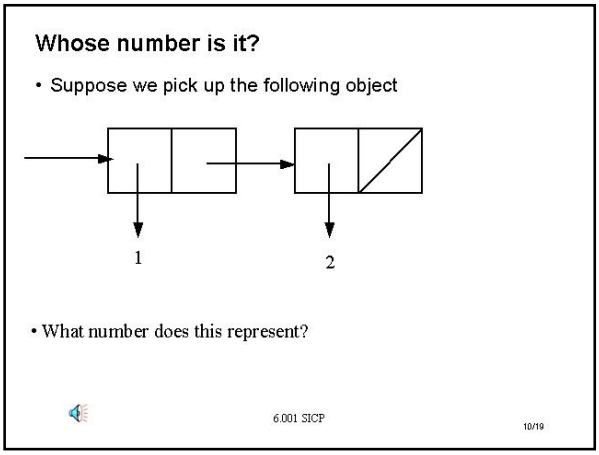


Slide 9.1.8
Completing Ernie's task is similar to Bert's. For the magnitude and angle selectors, we can just use the underlying list selectors to complete the contract. For selectors for Cartesian coordinates, we will need to select out the underlying parts, convert using some trigonometry to the appropriate values, and then return those values.
Notice that Ernie's implementation for complex numbers thus also meets the contract for such objects. All that has changed with respect to Bert's implementation is the choice of how to glue basic components together.

Slide 9.1.9
So far this sounds fine. We have two different implementations of complex numbers, one in Cartesian coordinates and one in polar coordinates, but both seem to satisfy the contract for the data abstraction. What's the big deal?
Well, suppose we find a complex number lying on the floor, such as the one shown...



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

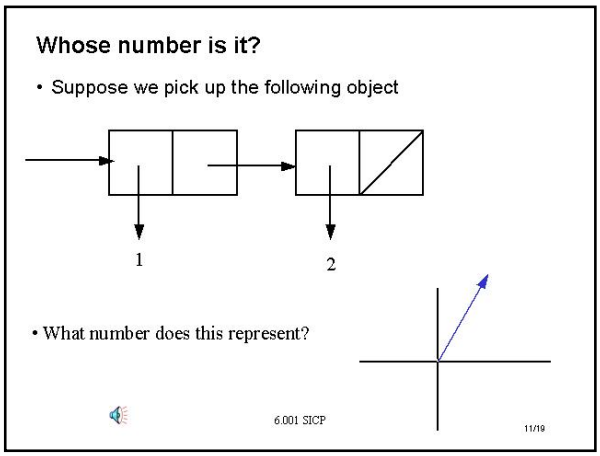


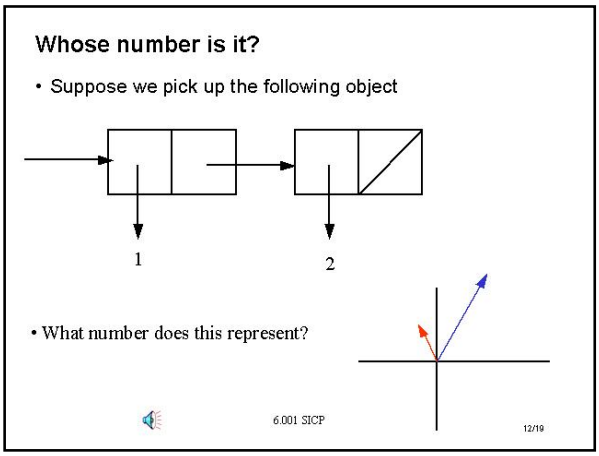
Slide 9.1.10

.. then here is the key question. What actual number does this object represent? This may sound odd, as after all, it is just a complex number. But suppose this is a complex number made by Bert. Then what number would this represent?

Slide 9.1.11

In that case, we know that the first part of this list represents the real part of the number, and the second part of the list represents the imaginary part. Thus, in this case, this number would correspond to the vector or point shown on the diagram.





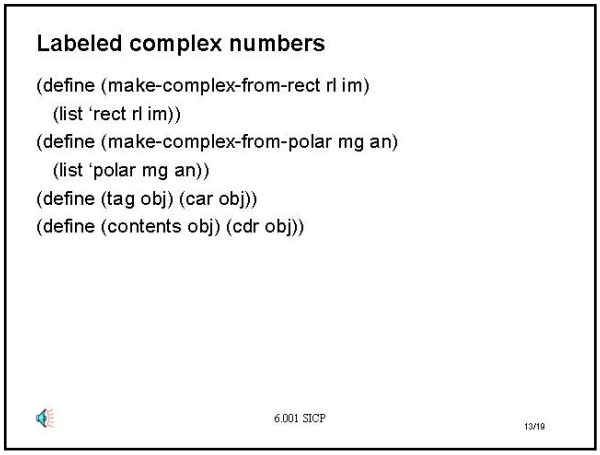
Slide 9.1.12

On the other hand, if this were a complex number made by Ernie, we know that the first part of the list is the magnitude and the second part of the list is the angle of the vector. In that case, this number represents the red vector or point shown on the diagram.

Thus, we have a problem. Depending on who made the complex number we found, we get a different answer to the question of what number this is. So how do we tell who made it? That is exactly the problem we are raising. Given what we have shown so far, we can't tell. Fortunately the solution is easy. Let's create "designer" complex numbers, let's have the designer of each kind of complex number sign his work on the back. That means we will have each creator of complex numbers but a label on the object that either says this is a "Bert" or Cartesian number, or this is an "Ernie" or polar number.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

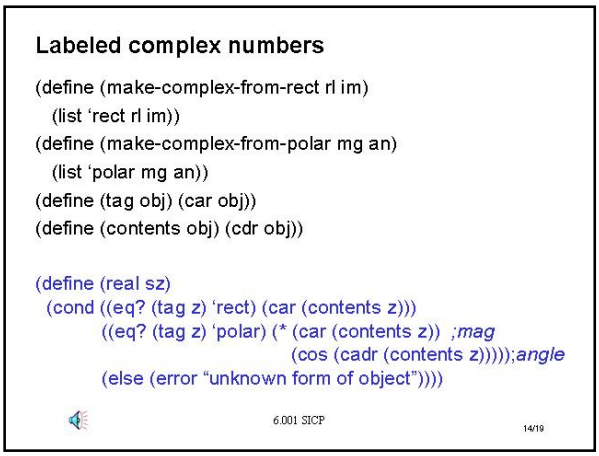
Slide 9.1.13



How do we add a label to our complex numbers? Let's just glue it onto the front. We can change our constructor to have the following behavior. Given two parts, we will just glue them together into a list. But if the parts represent real and imaginary components, we will put a symbolic label at the front to tell us this, while if the parts are magnitude and angle, we will still just glue them together but we'll put a different label at the front to tell us this.

What else do I need? I'll want a selector to pull the labels off of these new abstractions, as well as a selector to get out the actual **contents** of the abstraction. Notice the use of `cdr` in this case to get the remaining list of elements from the representation.

Note that in this case we are not doing any work to convert representations. We just glue the parts together, and put a label on it so we know what those parts mean.



Slide 9.1.14

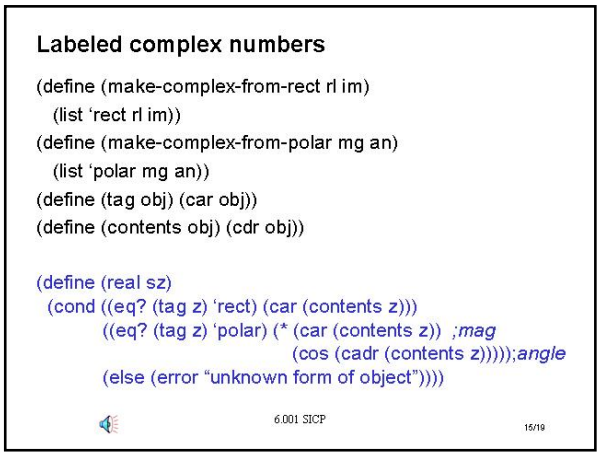
To make sure my contract holds, I'll need to adjust my selectors. Here is where I am going to bury the work I just saved in constructing numbers. I will have a single selector `real` that works with numbers represented in either form. This selector will first rip the tag off of the number to see who made it. Notice how we use `eq?` to test equality of symbols, and how we use the selector `tag` to get out the tag, and how we use the quoted symbols to represent the things to check against the tag.

If this is in fact a Cartesian number, we use `contents` to get everything but the tag, then can use `car` to get the real part, since that is where it is stored in this representation, just as Bert did it.

If this is a polar number, then we have to do the work to convert the underlying representation from polar form to the real part. In this case, `contents` will get the actual representation, and the `car` of that list we know is the magnitude of the vector. Similarly the `cadr` gets us the angle, and we can then do the trigonometry to convert to the real part.

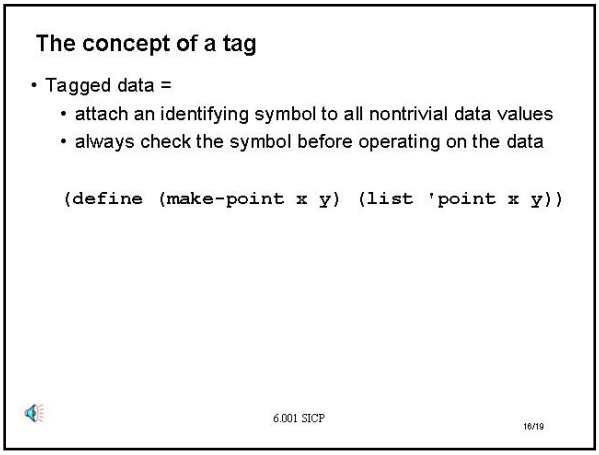
Thus in this case, my constructor just glues pieces together, with an appropriate label. My selectors use the label to tell me how to interpret the pieces, and thus what additional work I may have to do to convert those pieces to the desired value.

Slide 9.1.15



Note the key point here. It's not that I am dealing with complex numbers, they simply provide the motivation. The key point is that now I can use any of the procedures I wrote to manipulate complex numbers on any kind of complex number. Thus, independent of whether a complex number is actually glued together in Cartesian or polar form, the same procedure applies. To make this happen, I use tags (or types) to tell the selectors which pieces to pull out, and what to do to convert the pieces to the right form. And these tags are exactly what support the idea of having different versions of the same abstraction be handled by the same procedures.

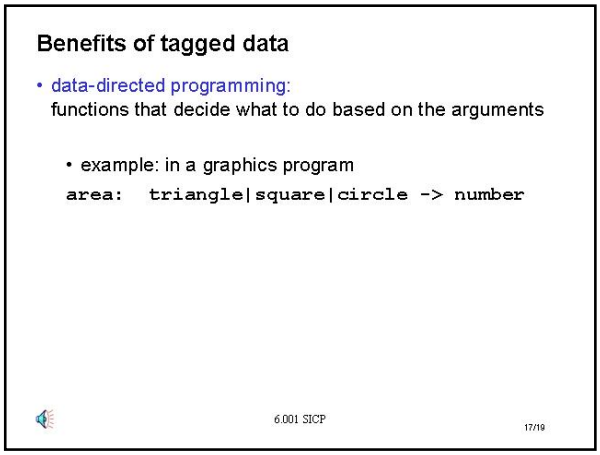
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.1.16

This simple example thus motivates the key idea we are going to explore in this lecture: by putting tags on data structures, we can identify the right operations to apply to that data structure. In fact, a careful programmer would always put tags on data structures, exactly to provide the flexibility to extend his or her system, and to provide a means verifying that correct operations are being applied to data.

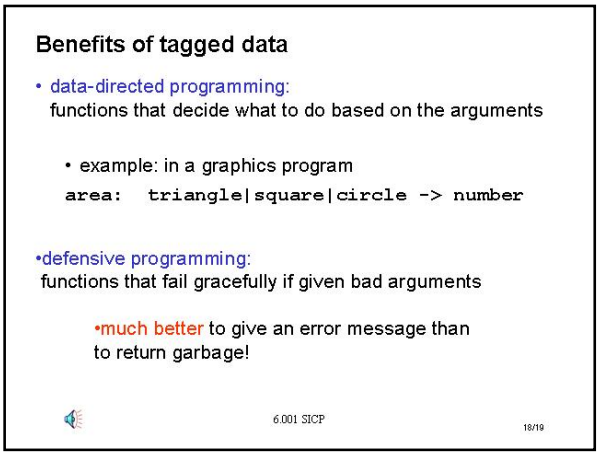
Thus, **tagged data** now refers to the concept of attaching an identifying label to all non-trivial data types in my system. By convention, we will try to always check the label on the data structure before applying any procedures to manipulate those structures.



Slide 9.1.17

As we will see, there are two key reasons for wanting to use tagged data. The first is that it makes available to use the powerful idea of **data directed programming**. Here, the idea is to let the type of an object direct the procedure to the right method to apply to that type of object. This means that our style will be to write procedures that look at the type of the argument, and use that information to apply a procedure specifically tuned to that type of object. This, in fact, is exactly what we just did with complex numbers. In that case, the selectors used the data type to direct the system to the right method.

As another example, suppose you are writing a graphics program, and want to be able to compute the area of a figure. Rather than writing one giant procedure to do this for all types of figures, we could let the type of the figure (a triangle, a square, some other form) direct the procedure to a subprocedure specifically designed for that type of figure. Such an approach leads to very modular code, which is much easier to modify and maintain.

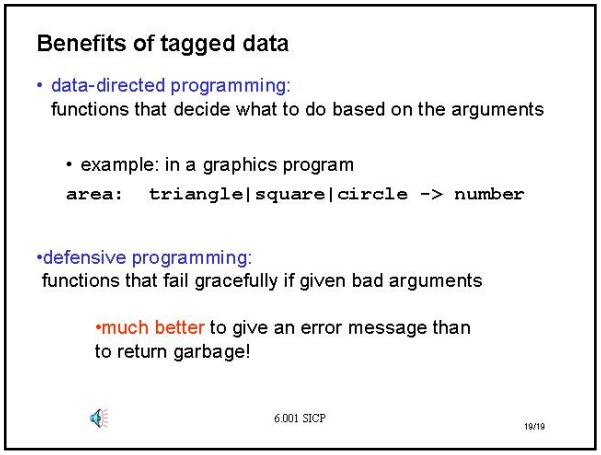


Slide 9.1.18

The second key reason is that this approach allows us to practice **defensive programming**. We want to be careful to ensure that we don't have unwarranted assumptions about inputs to our procedures, and in particular, we want our procedures to fail gracefully when they unexpected receive inputs of the wrong type. As we saw in the previous example, when we created a selector for a complex number, we did exactly that. I checked for specific types of objects, specifying the method to use, but if I did not recognize the type of the object as something I was set to handle, I returned an error message with appropriate information. The point is that by handling unknown types here, rather than assuming something about a data object, we catch errors before they can propagate. It is much harder to debug code if the error doesn't show up until several stages of additional processing have been applied to that incorrect data type.

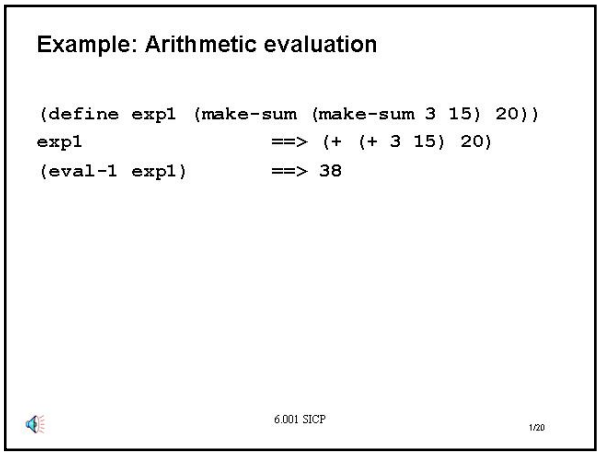
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 9.1.19
To summarize, we have introduced the idea of tagged data types, and used that to consider clean ways to modularize systems. We now want to explore in more detail how using tagged data makes it easier to build large systems.



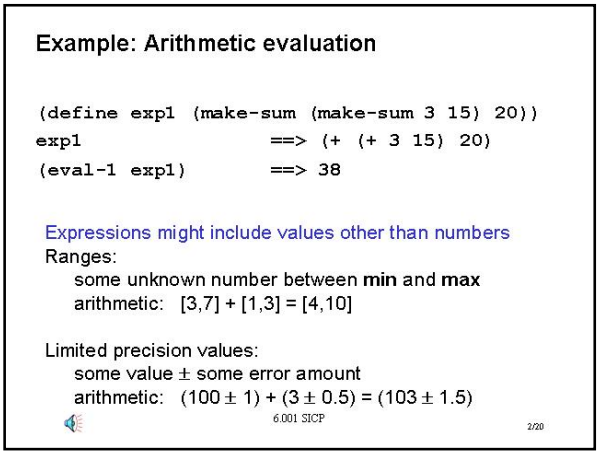
6.001 Notes: Section 9.2

Slide 9.2.1
To show how to use these two ideas: data directed programming and defensive programming, the rest of this lecture is going to consider an extended example. Because this example involves a fair amount of code, it is important to keep in mind the key points being illustrated, so you don't lose sight of them amidst all the code fragments. To make this easier, we are going to incremental build new versions of the example on top of simpler versions, thus highlight key ideas and changes.



The example we are going to use to illustrate the ideas of data directed programming and defensive programming is a system to manipulate **arithmetic expressions**. This will be similar to our example of symbolic derivatives, but now applying to more general expressions. So what does this mean? Not only do I want to be able to create symbolic arithmetic expressions, using appropriate constructors, I also want to be able to reduce those expressions to simpler forms, whenever possible. Thus, I would like to create symbolic expressions, such as `exp1`, as shown. Thus the value of the `exp1` will be the actual expression. And, I want to create a system that can **evaluate** expressions such as `exp1`, that is, reduce this expression to a simpler form, in this case, the simpler expression `38`.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.2.2

Building a system to evaluate expressions such as those just shown would be interesting in its own right, but I would like to add more to my system. In particular, I want my system to not only simplify standard arithmetic expressions, but also expressions that involve numbers that are only known to lie within a particular range. For example, I want my system to be able to simplify expressions where the numbers are only known to lie between specific bounds of uncertainty. In the example shown, I may only know that one number lies between 3 and 7, and another number lies between 1 and 3, but I still want my system to be able to simplify an addition involving these numbers, whose result I know must lie between 4 and 10.

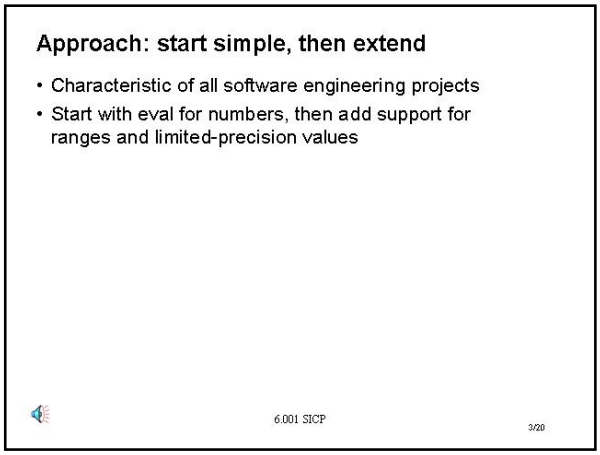
Moreover, I would like my system to be able to deal with expressions involving data from scientific experiments, in which I know the ostensible value of a number and a specified range of precision, that is, that the true value may lie within some plus/minus range of the ostensible value. In the example shown, I want my system to be able to add together expressions where I know one number is 100 plus/minus 1 and the other number is 3 plus/minus 0.5, and have the system deduce that the sum should be 103 plus/minus 1.5.

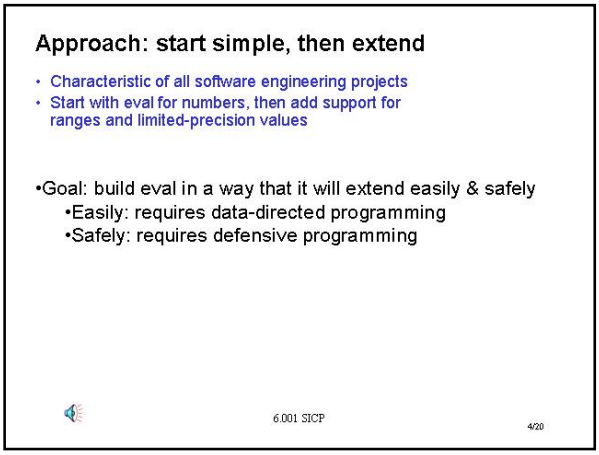
Thus, my goal is to build an arithmetic evaluation system that reduces arithmetic expressions, consisting of a mix of symbols and numbers, to simplest form; where the expressions may be standard numbers, ranges of values or limited precision values.

Slide 9.2.3

The basic approach we are going to take is to start with easy things first, an obvious thing to do! In our case, we will first build a system to handle normal numbers, and then we will look at how to build on top of that to handle other expressions, like ranges and limited precision values.

This does sound obvious, but it is important to stress that this is an important characteristic of a well-designed software engineering project, (and one that all too often is not followed, even in commercial systems). It is almost always easier to extend a base system, than to try to do the whole thing at once. This is an important lesson to learn, and one that hopefully you will see in this extended example.





Slide 9.2.4

One of the things to watch for as we go through this exercise is to notice how, by doing the development in stages, we make extensions to the system much easier to conceptualize. Indeed, our goal is to build our simple evaluator so that extensions are both easy and safe.

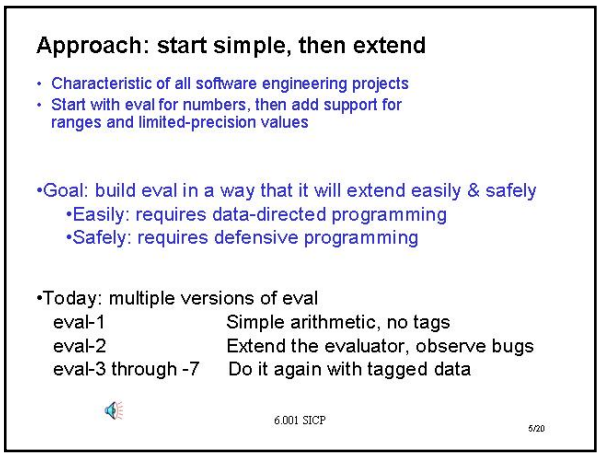
So what does that mean? To allow for easy extensions, we will need to utilize tools from data-directed programming. That is, if we use data-directed programming tools (e.g. tagged data, dispatch to methods based on tag types), we will see it is easy to add new types of expressions to our system (create a new structure, an appropriate tag, and a dispatch to a method to handle that new type).

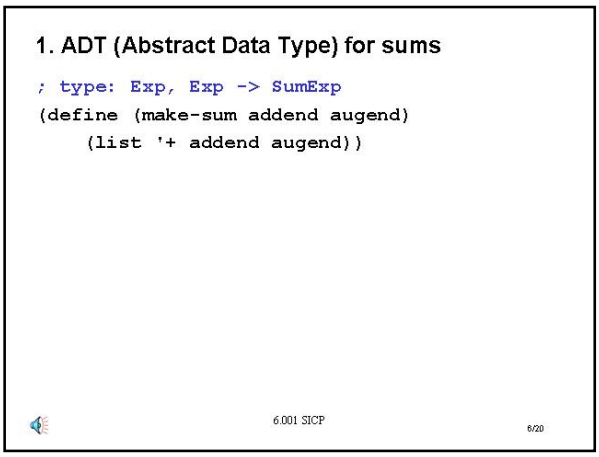
To allow for safe extensions, we will need to use methods from defensive programming. This will require explicit

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

tests for types, error checks to catch unexpected types of arguments, and disciplined use of tags to tell use how to handle expressions rather than assuming an expression is of a particular type.
Thus, as we go through this exercise, watch how both of these themes guide the development of the system.

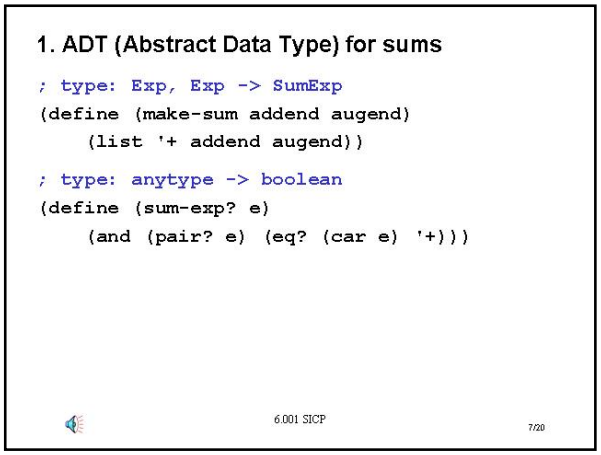
Slide 9.2.5
Here is our plan to accomplish this. We will first build an evaluator to simplify expressions involving normal numbers. Our second version will extend this base version in an obvious manner. We will see that this obvious extension is actually flawed, and we will use the insights from its failure to create a correct extension, using a series of versions of the evaluator. This will be a cycle in which we extend the system, observe its behavior, and use the observation to guide the next extension. Thus this series of extensions will highlight how data-directed programming and defensive programming intertwine to guide the development of a complex system.



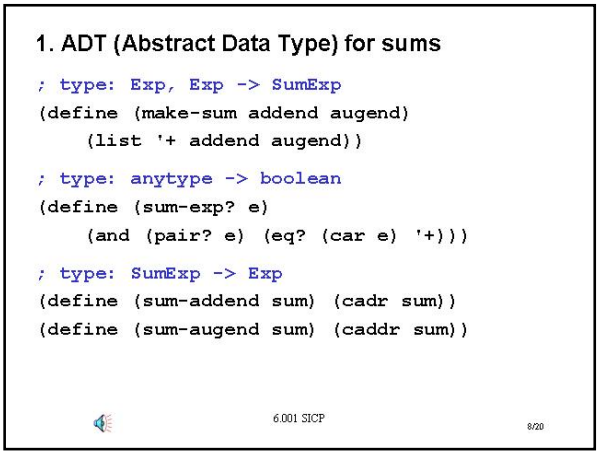


Slide 9.2.6
Let's start with our simple "sum" expressions. First we build a constructor, as shown. Notice the type of this constructor. It takes in any two expressions, and returns a `SumExp`, something of a more particular type, which is identified by the symbol `+` at the front of the expression. This symbol serves as the tag for the expression, and we are taking advantage of the fact that the operator itself identifies the type of expression. Thus this constructor creates a tagged data type of particular type "sum", by gluing together the subordinate expressions with the tag.

Slide 9.2.7
Associated with this object type will be a predicate to detect instances of such objects. `Sum-exp?` takes in an expression of any type, and returns a Boolean value, that value indicating whether the input expression is a sum. Check out the body of this procedure. We first check to see that the expression is a pair (not that `and` evaluates its arguments in a left to right order, and stops as soon as one of its arguments returns a `false` value). If it is not, we stop, and return `false`. If it is a pair, then we can safely take the `car` of it, and check for the symbol `+`. Note that this is an instance of **defensive programming**.
Also note how we are using the tag to identify the type of object.



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

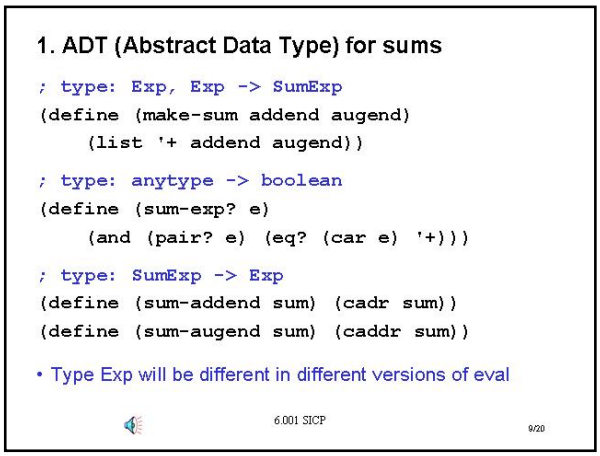


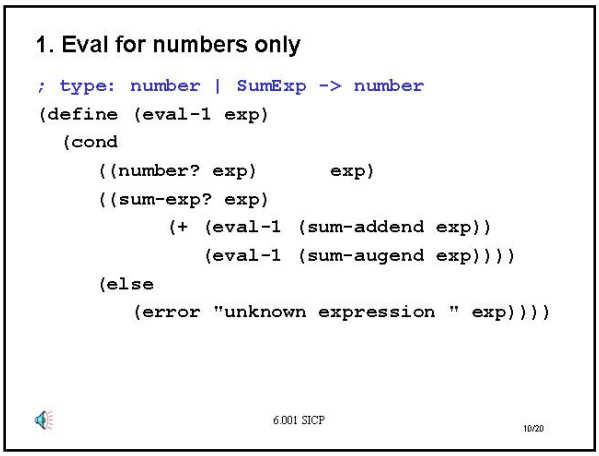
Slide 9.2.8

And of course we will need selectors for the data objects. The type definitions for these procedures are that they convert `SumExp's to Exp's.` Note that we can assume that the selectors are given `SumExp's` as input, that is, we have used defensive programming to check the type of the object, before applying the selector. This means that we can safely use `cadr and caddr` rather than first checking that the argument is a list, since the procedure already knows it is getting a `SumExp` as input

Slide 9.2.9

The last thing to keep in mind is that here we are just dealing with sums, so the type of expression returned by the selectors will just be "sums". But obviously, as we add other kinds of expressions to our system, we will need to the tags to help use direct the system to the correct subprocedure.





Slide 9.2.10

Given this starting point, it is straightforward to implement an evaluator for reducing "sums" of numbers and variables. First, notice the type of this procedure. It takes as input either a number or a `SumExp,` which we know is a list of three things, the first of which is the tag +, the other two of which are themselves numbers or sums. Our desired output of this procedure is a number, that is, we want it to reduce the input expression to a simple form.

So how do we do this? Well, if the expression is just a number, there is nothing to do, as it is already in simple form. Notice what this is doing. We are using a data-directed style to check expression types, looking for the base or primitive types first. In this case, we use the built-in Scheme predicate for numbers to decide if the expression is already in simple form.

If it is not a number, we then use defensive programming to check if the expression is a sum. If it is a sum expression, then we use the selectors to get out the pieces, and recursively apply the evaluator to those expressions to reduce them to simplest form. Note that we are safe in applying the selectors here, since the defensive programming style has ensured that the expression is a sum.

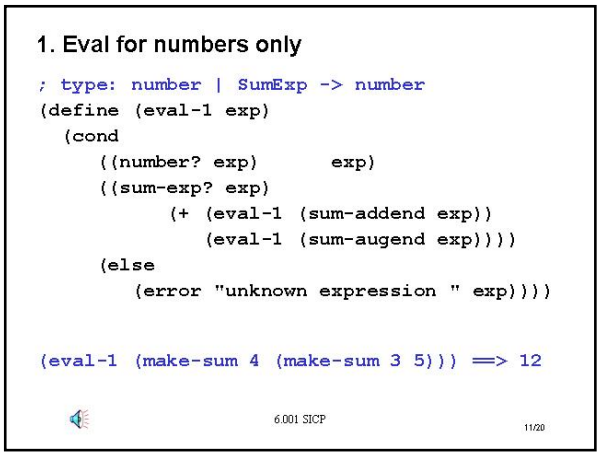
This recursive application of the evaluator will allow us to deal with expressions whose subexpressions are themselves sums, and so on. If we have built `eval-1` correctly, then we are guaranteed that once these subexpressions have been processed, no matter how deep a nesting of expressions they contain, the result will be a number. Thus we can then add the results of evaluating to the parts of the sum, and return a number as the final value.

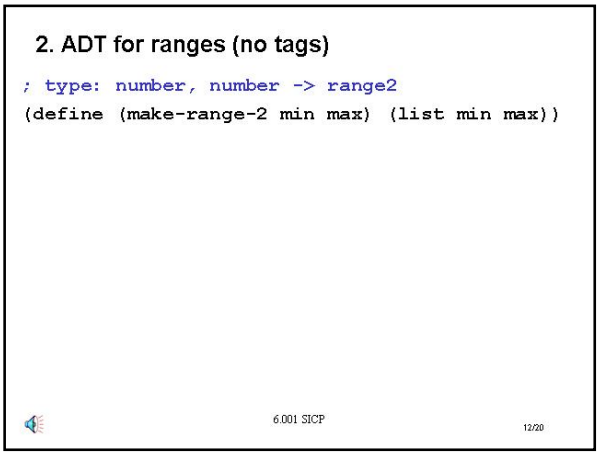
Finally, notice the defensive programming. If the argument is not a type that we know how to handle, we exit with an error. We don't just assume that if the expression is not a number, it must be a sum, we check explicitly. This

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

way, if some other part of the system contains a bug, we will be able to spot it and isolate the bug, rather than propagating that error further down the processing chain.
Also notice the form of the procedure. We have a **base case** that deals with primitive expressions, and we have a recursive case. In the latter, we pull out the pieces using selectors, reduce the problem to a simpler version of the same problem, then combine the results using simple operations.

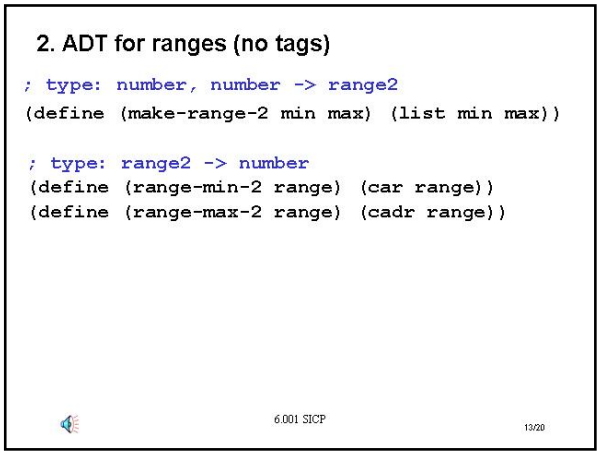
Slide 9.2.11
If we apply this procedure to a sum that includes as subexpressions both numbers and sums, it correctly unwinds the evaluation. You should be able to trace this by applying the substitution model.
Given this system, let's see what happens as we try to extend it to handle other kinds of expressions.



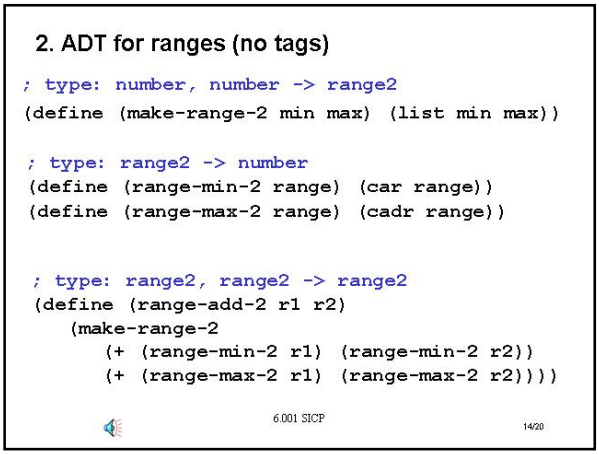


Slide 9.2.12
So, let's try the obvious extension. Let's add **ranges** to our system. This means we will need a data abstraction for ranges, and to do it the "dumb" way, we'll do this without using explicit tags. Thus, the obvious way to represent a range is to simply glue together the `min` and `max` values of the range, in a list.

Slide 9.2.13
Given that choice for a representation, the implementation of the selectors is easy. We simply take one of these ranges, represented as a list, and pull out the right piece. This simply uses the abstraction contract for lists to create selectors that meet the desired contract for ranges.



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



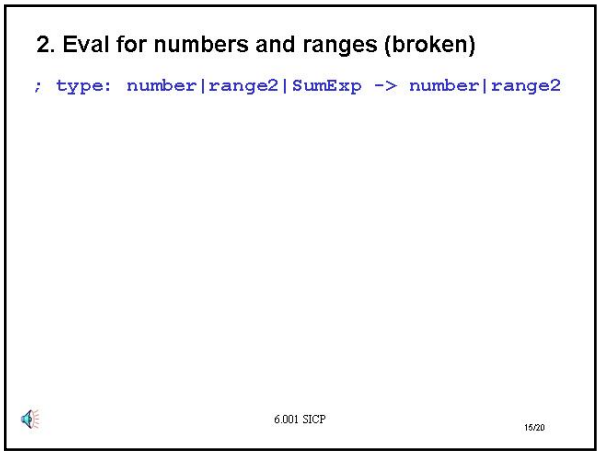
Slide 9.2.14

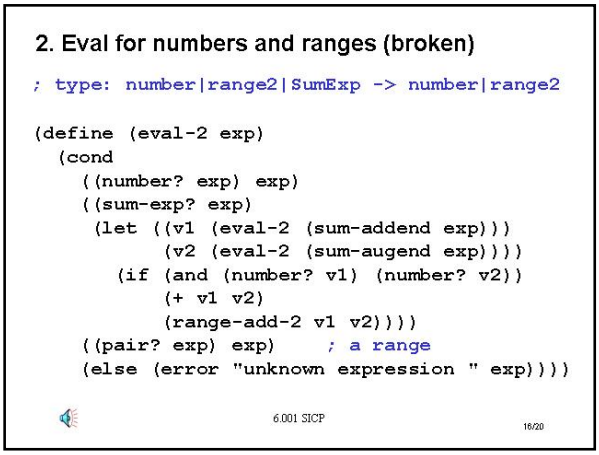
Then building a procedure to add ranges is just a matter of using the selectors to pull out the min and max values of the two input ranges, then using normal addition to create the new min and max of the sum, and finally using the constructor to glue those new extreme values together into the new range.

Note what we do here. We know the inputs are ranges; hence we can safely apply the selectors. By their type definitions, we know that they return normal numbers, so we are safe in applying + to them. Then given two numbers for the new min and max, we can use the constructor to create a new range, since the input types match it's type definition. Thus, we meet the contract for the type definition of `range-add-2`.

Slide 9.2.15

Using this representation for ranges, we can now build our second version of an evaluator, one that is intended to deal with numbers and ranges. Notice our desired type definition for this procedure. The input argument should be a number, a range or a sum expression, and it should return either a number or a range, since those are our two primitive forms of expressions.





Slide 9.2.16

Here is the desired procedure. Look at the structure of this second evaluator. Note the form. As before, it first checks to see if the expression is a number, in which case, we are done, and can just return the number.

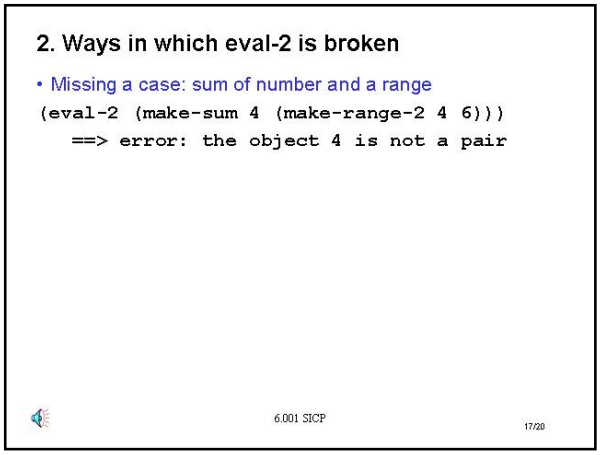
If it is a **`sum`** expression, we will first recursively simplify the two subexpressions. Then, we will try to be clever. We'll check to see if both values are numbers, in which case we will just add them together the normal way. Otherwise we will add the two values together using a procedure designed to add ranges.

Finally, we will check to see if the expression is a pair. Since we already know it is not a sum expression, if it is a pair, then it must be one of our ranges, and we will just return the range.

This looks okay, right? ... well, let's check it out ...

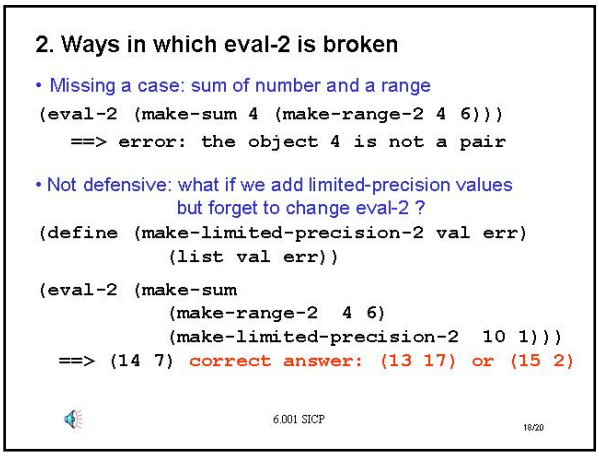
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 9.2.17



... as you have probably already guessed, this in fact is "not right". Here are some ways in which this system does not behave correctly. Consider the first example. If I try to evaluate a sum made up of a number and a range, I get into trouble because in fact I haven't allowed for all possible cases in my procedure. In particular, why does this fail? Well, the evaluator will try to simplify this sum, and in doing so will first check to see if the two parts are numbers. They are not, so it assumes that both expressions are ranges, and tries to add them together as ranges. This finally breaks down when we try to take the `car` of the first expression.

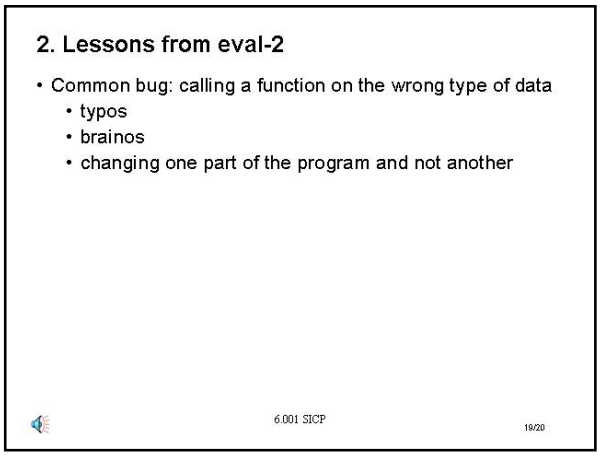
So why is the system failing in this way? Largely this is because our code is making assumptions about how data structures are represented. In particular, we are missing a case in which we have a sum of a number and a range, which we need to deal with.



Slide 9.2.18

Here is the second problem. We weren't really defensive in our programming. Suppose we add a limited precision data type to our system, as shown. Then notice that our evaluator will actually produce an answer, but an incorrect one. It will treat anything that is a pair as a range, and add the parts together as if they represented minimum and maximum values, rather than a value and an uncertainty. Of course, in a defensive approach, the system should have detected that this is not a data type that it can handle and alert us. The problem here is that we assume that pairs represent ranges, without putting a tag on them, and thus end up using the same representation for other types of objects.

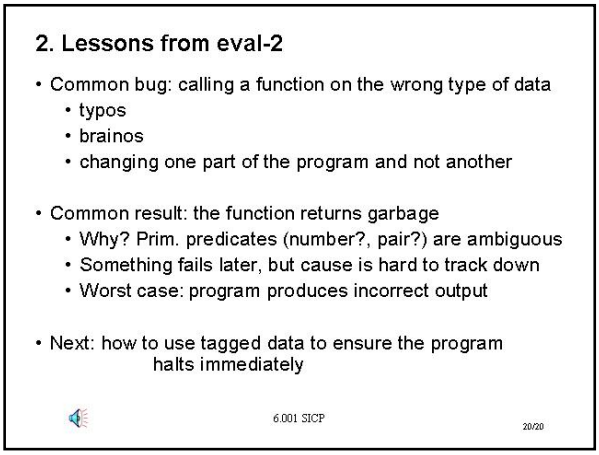
Slide 9.2.19



Even though this is a simple example, it nicely illustrates some of the lessons to be learned in building complex systems.

The first is what we just observed. A common bug arises in calling a function on the wrong type of data, either because we made a coding error, because our reasoning was flawed or overlooked a possibility, or because we change one piece of code and forget to change the corresponding pieces elsewhere in the system.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



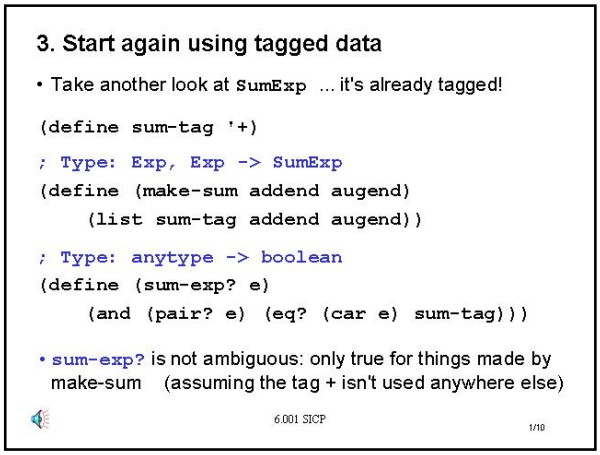
Slide 9.2.20

The result is that the system produces useless values. It is particularly a problem when it does not hit an error, but returns a value that it believes (incorrectly) to be correct. As we saw, the system can sometimes produce an answer that looks like a legal response, but is not. In this case the system may not fail until much later in the process (or may even return the answer as a final result), where it is much harder to track down the cause of the failure (or possible to miss it altogether).

In this case, our underlying problem is that we are only loosely using the data types to represent our objects. We really need to be disciplined in tagging all our data types, and not relying on underlying representations to be uniquely associated with data types.

6.001 Notes: Section 9.3

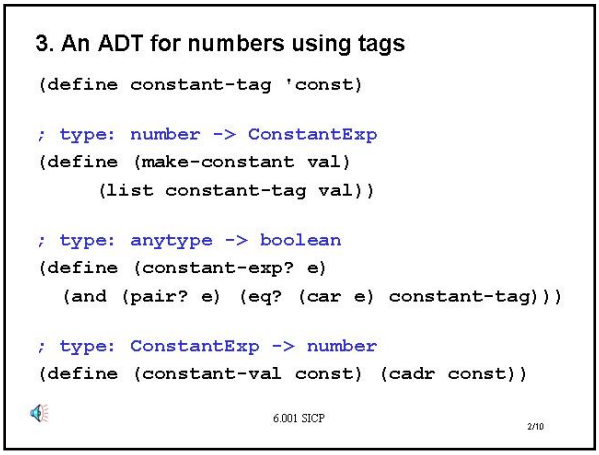
Slide 9.3.1



So let's go back and look at our sum expressions, with this idea of using tags. In fact, our sum expressions are already tagged. The first subexpression identifies the type, so let's pull this out explicitly and give it a name. Thus we will restructure our data abstraction to isolate the tag and its use. We will create an explicit label, and use it everywhere that we previously used the implicit label of `+`. It may seem a bit silly that we are just using `+` as the label, but giving it another name, but the advantage is that now if we decide to change to another label, we need only change the definition for `sum-tag` which is just one change in the code, rather than finding all the places where we rely on that tag and changing them. In other words, we isolate the use of the tag from its actual value.

The key point is that now the predicate for testing if an expression is a "sum" is not unambiguous. We check for equality of the tag with the value of the special symbol `sum-tag`. So long as no other constructor uses `+` as a tag, only things constructed with `make-sum` will pass the predicate `sum-exp?`.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.3.2

Given that we are being more disciplined about our use of tags, we need to go back and label our primitive expressions as well. As in the previous case, we will create a tag to identify constants, and give it a unique name. By using this name, we isolate the actual value of the tag from the use of the tag, which again means that if we decide to change the specific tag, we can do so with one change of code, rather than having to find all the myriad places where we use the tag.

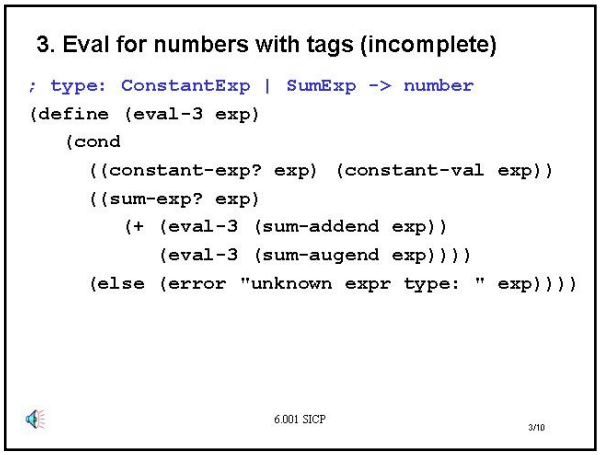
What else do we need? Now, we need an explicit constructor for constants. In the earlier version, we just used numbers directly. Here, we explicitly construct a tagged abstract data type, by gluing the label onto the front of the actual number.

Of course, we can now define a predicate for testing whether an expression is a `constant`. Notice the defensive programming, in which I first check that the expression is a pair, before trying to get out a piece of a cons pair. The actual check is whether the tag of the expression is `eq?` to the **value** of the name `constant-tag`.

Since I now have a tagged object, I will also need a selector to get the actual value of the constant.

Thus, this data abstraction is very similar to our earlier ones. We have a tag at the front, we have a constructor for explicitly making instances of the object, we have a careful check for the type of expression, and a selector that relies on the predicate having already verified the correct type so that it can directly pull out the right piece of the structure.

Slide 9.3.3



Given this, we can now restructure our evaluator, so here is the third version.

First, notice the new type. Here we explicitly acknowledge that the input expression is either a constant expression or a sum expression, and that the result is a number.

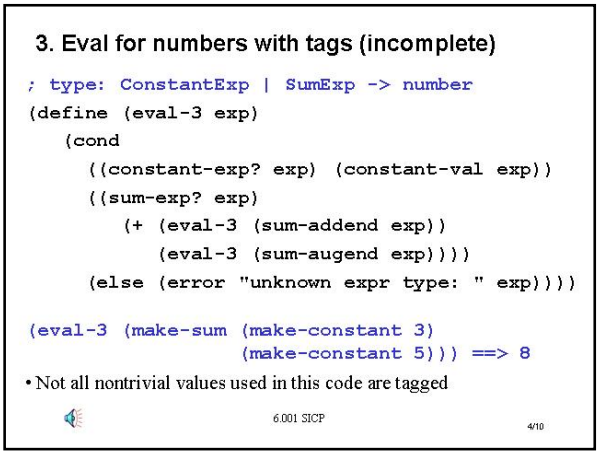
Notice how the evaluator is structured. In this case, it explicitly checks the tag on the expression for the type. Thus, it doesn't rely on underlying scheme types (e.g. `number?`) to identify types, it looks for an explicit tag.

This leads to a very nice overall structure, with a case for each kind of object in the system, and a failsafe case to be defensive in our programming. Thus, if the system gets something other than an expression constructed by one of our constructors, it will tell us immediately, rather than assuming that there is a default type of expression, as we did in the previous case.

If the expression is a constant, we can just return the number associated with that data type.

If the expression is a sum, we get out the pieces, evaluate them, then add those two numbers together to return a single number as the result.

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.3.4

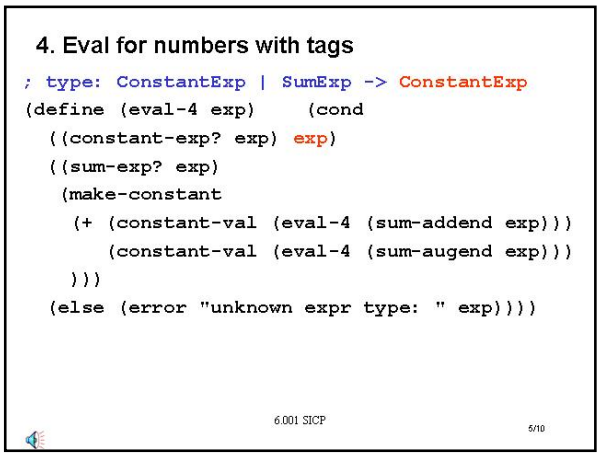
This looks nice, right? In fact, it does seem to have the right form. But we have still made a careless assumption.

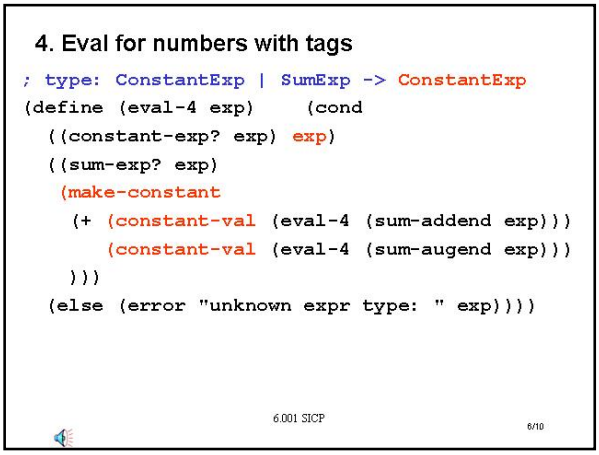
If we construct a sum of two constants (3 and 5) and evaluate it, we get out the number 8. This meets our type definition, but think about what happens if we were to use this expression inside a larger sum. In that case, we would end up trying to evaluate a sum with an untagged object (this value) to a tagged object, and we would hit an error. So we are closer, but we still are not there.

Slide 9.3.5

Fortunately, the fix is easy. We need to change the type characteristics to return a tagged object, that is, a **constant** rather than a **number**. Or, if you prefer, we need to ensure that we put the tag on the number before returning, because this might only be a subpart of some other expression, and the inputs to the evaluator assume tagged expressions.

Thus, our new and improved evaluator treats constants the same as before, except that now we return the fully tagged expression, not the value. Thus, we meet the new type characteristics, by returning a tagged expression.





Slide 9.3.6

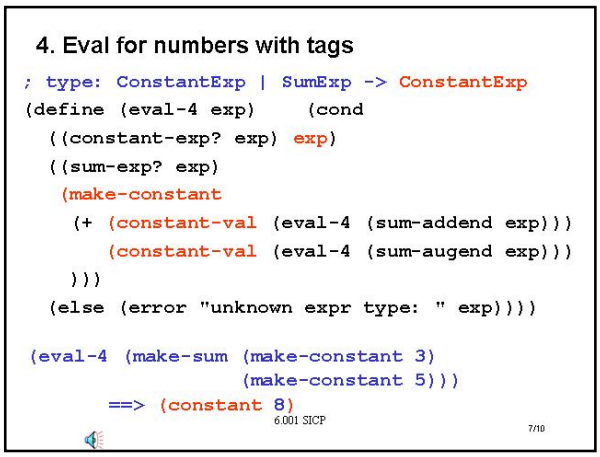
For sums, notice what we do. As before, we use the selectors to pull out the subexpressions. We then recursively apply `eval` to them, to get the reduced value. By our new type contract, this will return two **tagged** constants, so we need to extract out the values of each, add them using normal addition, then meet the type contract by attaching a tag to the result.

This looks like more work to code, but the result is a much cleaner implementation. Notice how in both cases we are now guaranteed to get a **constant** expression as the result, not a number.

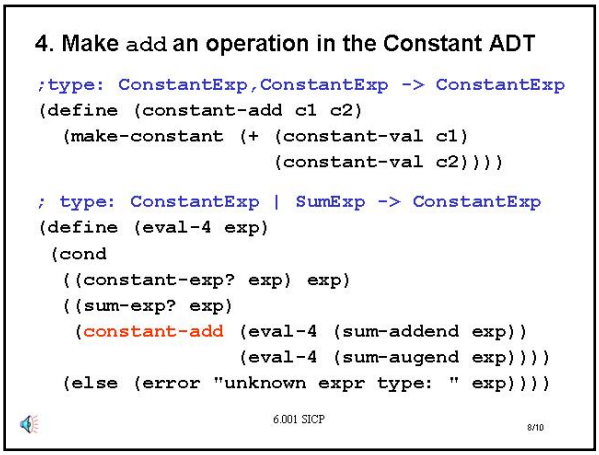
Slide 9.3.7

And now when we try this, we get back a labeled object, in this case the constant, 8, not the number, 8. This satisfies our new type contract, and means that if this were a subexpression in a larger expression, the returned value would be something that we could then use for further reduction, since the type tag would direct this to the right method.

Notice how this version of the evaluator has the property that every case in the dispatch procedure assumes the input is some tagged structure, and does not make an implicit assumption about types of objects. Further, each case returns a tagged object as a value.

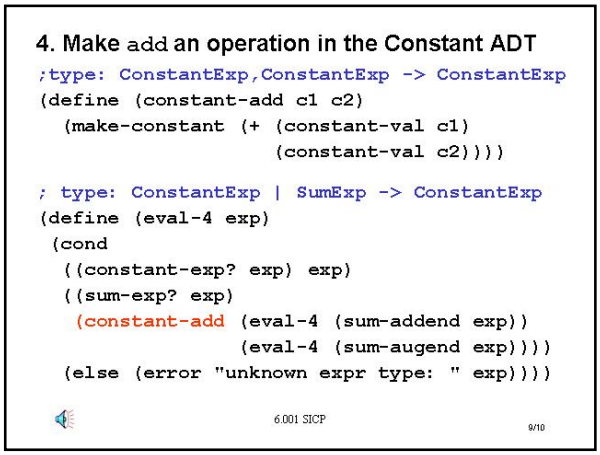


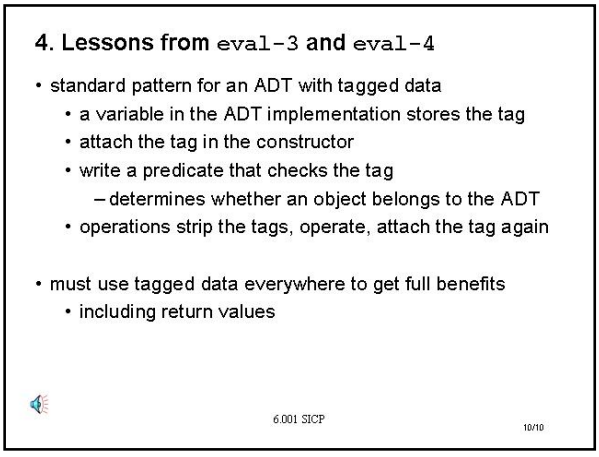
6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.3.8
In principle, this is a good thing to do. But it is not as clean a piece of code, as we would like.
Notice that we have actually intertwined operations on data types inside our procedure. Let's fix that. Let's pull that operation outside to be a part of the data abstraction, thus allowing our evaluation mechanism to be much cleaner, just focusing on operations on abstractions, rather than directly manipulating them.
Thus, we have our procedure for adding constants use the selectors to get out the parts, do the addition, and then remake a tagged data structure. This is much better, as now the procedure for evaluation only involves procedures that manipulate abstract objects, and all work inside the abstraction is isolated inside separate procedures.

Slide 9.3.9
Now our evaluator has a consistent form. It uses predicates to check types of expressions. Other than the failsafe case, it dispatches to a method designed to handle each kind of expression. There are no data construction or data manipulation procedures exposed within the evaluator, other than the selectors to get out parts of an expression. All that work has been isolated within procedures designed for each expression type.





Slide 9.3.10
If we step back from the example, we can extract a set of messages that apply much more generally. Our standard pattern for manipulating abstract data types is detailed at the top of the slide. This pattern is something we will want to use frequently as we build other, more complex, systems.
Notice the discipline associated with this pattern. We **must** use tagged data everywhere within our system, including all returned values.
While this example may seem simple (and perhaps boring) you should examine it carefully to be sure you are comfortable with the inherent ideas. You will be using this approach often as you build your own systems, and this discipline supports safe and efficient programming.

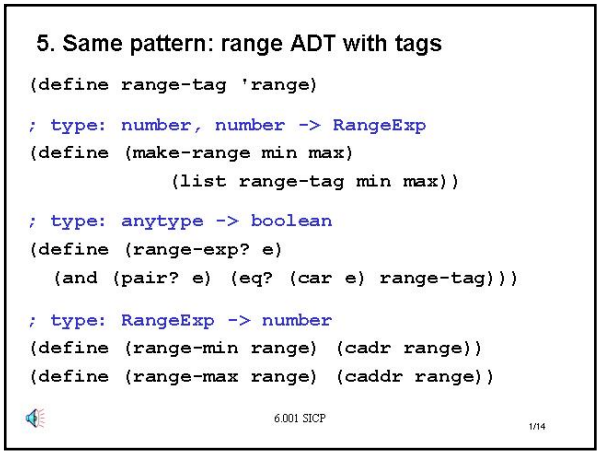
6.001 Notes: Section 9.4

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 9.4.1

Having learned how better to structure a tagged data system, we can go back to where we started. Remember that our goal was to build a system to handle sums of numbers, ranges and limited precision values. And our plan was to build a correct system for the simpler expressions, then extend it to handle a broader range of expressions.

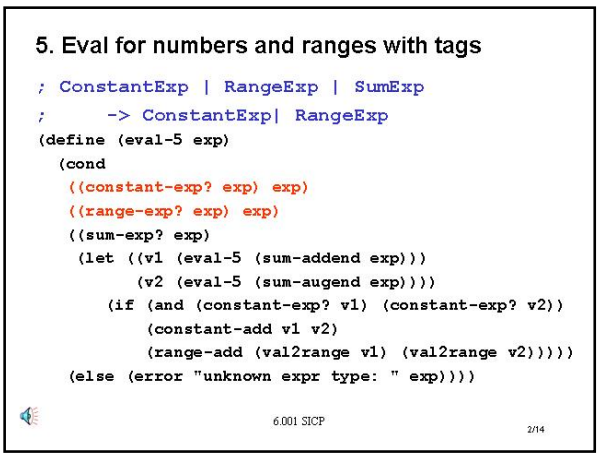
Having built a correct system for constants and sums of constants, we need to extend this to handle ranges. And to do this, we just apply the same ideas. This means create a variable name to hold the tag information (in this case `range-tag`).



This isolates the value of the tag from the use of the tag.

Our constructor, our predicate and our selectors look just like the previous system, with a tag attached in the right place.

All of this just mimics the structures we built for our last evaluator, but now applied to **ranges**.



Slide 9.4.2

Having added a **range** data structure, we can turn to incorporating such data types into our evaluator. First, what do we want in terms of behavior of the evaluator?

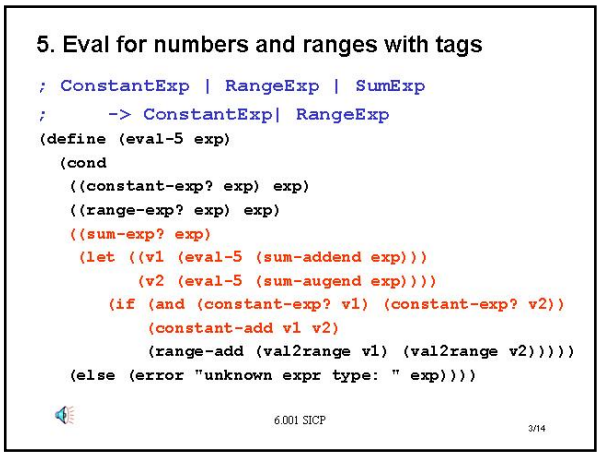
The argument to this procedure will now be a **constant**, a **range** or a **sum**, all of which will be appropriately labeled by a tag. We want the evaluator to produce either a constant, if the expression only involves constants, or a range, if the expression involves some combination of ranges and constants, or just ranges. Note that this implies a hierarchy of types, in which constants are subsumed by ranges.

Thus, if the expression is just a constant or a range, we are done, and we should simply return the tagged expression. Notice the two clauses to handle these primitive cases, both using data directed dispatch.

Slide 9.4.3

What about sums? In this case, we have to deal with the possibility that the parts of the sum could themselves be either constants, sums or ranges. Thus, we will first select out the parts of the sum, and evaluate those subexpressions. Depending on the types of values returned for each part, we can complete the simplification.

If both subexpressions evaluate to a constant, we can use a procedure that adds constants (which will select out the values, do normal addition, then glue a tag on the front and return an appropriate abstract data representation). Notice how this nicely separates out the data manipulation from the operation. All the actual manipulation of the data structures is isolated within `constant-add`.



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

5. Eval for numbers and ranges with tags

```
; ConstantExp | RangeExp | SumExp
;      -> ConstantExp| RangeExp
(define (eval-5 exp)
  (cond
   ((constant-exp? exp) exp)
   ((range-exp? exp) exp)
   ((sum-exp? exp)
    (let ((v1 (eval-5 (sum-addend exp)))
          (v2 (eval-5 (sum-augend exp))))
       (if (and (constant-exp? v1) (constant-exp? v2))
           (constant-add v1 v2)
           (range-add (val2range v1) (val2range v2)))))
   (else (error "unknown expr type: " exp))))
```

6.001 SICP 4/14

Slide 9.4.4

If the two parts are not both constants, then we need to return a range, since at least one of the values is a range. Here we need to be careful. In order to add two ranges, we need to insure that the inputs are ranges (that is what `range-add` requires in its type definition). Thus we will use another procedure that converts values to ranges to ensure that what is passed in to `range-add` is of the correct form. Of course, if a range is passed in to `val2range` we should just return that range.

Slide 9.4.5

As we did earlier, we can pull this piece outside of procedure and into the abstraction, thus simplifying the code. In this case, we create a predicate that tests whether the expression is one of our primitives. Note that this creates an implicit higher-level data abstraction, which absorbs two more primitive data abstractions into a common type.

6. Simplify eval with a data-directed add function

```
; ValueExp = ConstantExp | RangeExp
(define (value-exp? v)
  (or (constant-exp? v) (range-exp? v)))
```

6.001 SICP 5/14

6. Simplify eval with a data-directed add function

```
; ValueExp = ConstantExp | RangeExp
(define (value-exp? v)
  (or (constant-exp? v) (range-exp? v)))

; type: ValueExp, ValueExp -> ValueExp
(define (value-add-6 v1 v2)
 (if (and (constant-exp? v1) (constant-exp? v2))
     (constant-add v1 v2)
     (range-add (val2range v1) (val2range v2))))

; val2range: if argument is a range, return it
; else make the range [x x] from a constant x
```

6.001 SICP 6/14

Slide 9.4.6

... and for this higher order kind of data abstraction, we will create a procedure to add such data types together, by considering exactly the set of cases we did earlier.

Slide 9.4.7

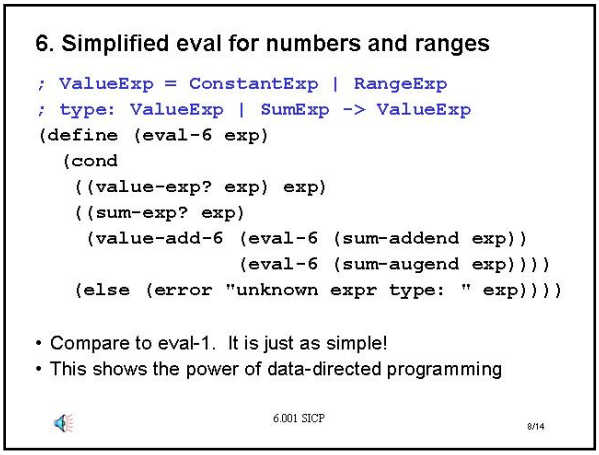
.. and here is why we want to do this. Now we have a nice, crisp, clean version of the evaluator. Notice the new type definition here, which uses our new data abstraction.

6. Simplified eval for numbers and ranges

```
; ValueExp = ConstantExp | RangeExp
; type: ValueExp | SumExp -> ValueExp
(define (eval-6 exp)
  (cond
   ((value-exp? exp) exp)
   ((sum-exp? exp)
    (value-add-6 (eval-6 (sum-addend exp))
                 (eval-6 (sum-augend exp))))
   (else (error "unknown expr type: " exp))))
```

6.001 SICP 7/14

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.4.8

Notice what the evaluator actually does. There is a simple dispatch, one for a value expression, one for a sum expression. These dispatches are again based on using the tags on the expressions. For value expressions, we just return the expression, as there is nothing more to simplify.

For a sum expression, we use the same pattern as before. We select out the subpieces, recursively evaluate them to get simpler forms, then recombine the results into an abstract data type.

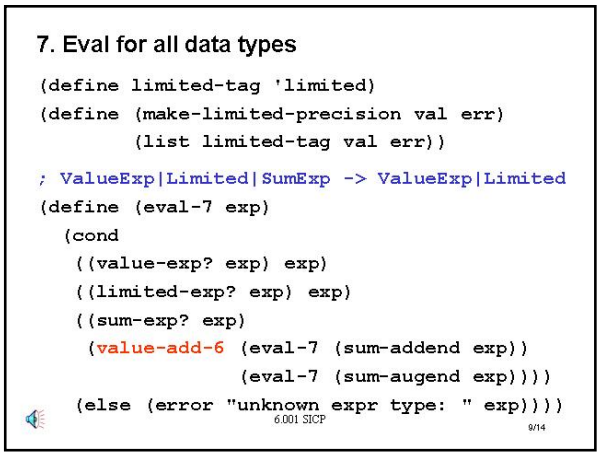
Notice the nice structure of this code. We have a base case for the primitive expressions. We have a recursive case, in which we reduce the problem to simpler versions of the same problem, plus a simple method for combining the results.

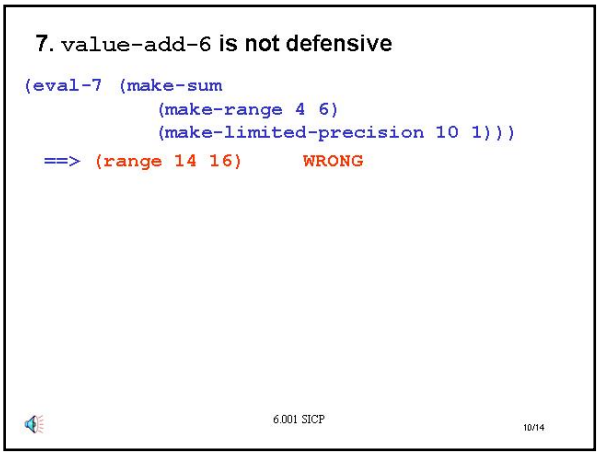
Perhaps most importantly, compare this to our first evaluator. It has exactly the same simple structure, but because of the disciplined way in which we have used data types and abstractions, we can handle a much bigger range of expressions.

What makes this possible is the separation of tag use from tag value, the use of tags to direct the processing, and defensive programming to handle unexpected types. Also notice how we have used the type definitions of the various procedures to help us reason about what each procedure should do.

Slide 9.4.9

To add in the **limited precision values** we will use exactly the same approach. Since we are adding a new simple type of expression, we will add a new base case to our evaluator to handle this type. In principle, we might think that this is enough, but what happens if the two pieces of a sum are now combined?





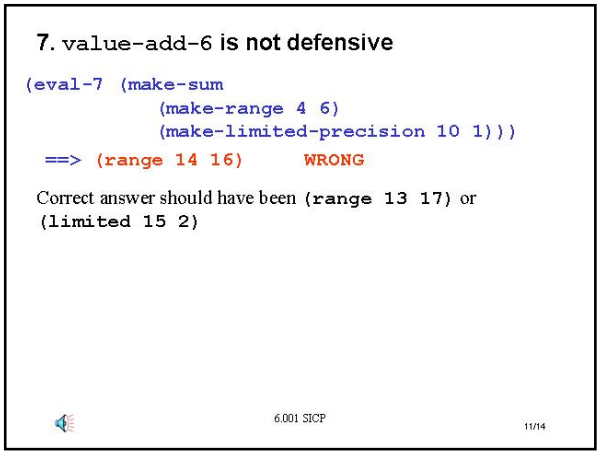
Slide 9.4.10

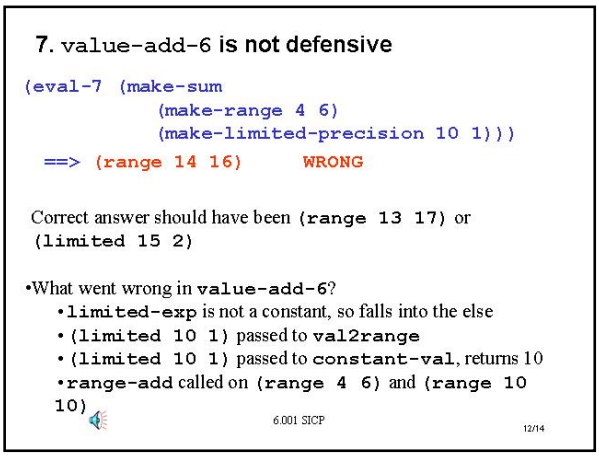
As you have probably already deduced, we have to be careful here. In particular, we need to exercise defensive programming. To see this, consider the example shown in which I make a sum of a range expression and a limited precision expression. If I evaluate this, that is, if I reduce it to simpler terms, I get back a **range** from 14 to 16. But that is **incorrect!**

6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.

Slide 9.4.11

The right answer should either be the **range** from 13 to 17 or the **limited precision value** of 15 plus/minus 2. Instead, we got this hybrid mix. Why?





Slide 9.4.12

.. because we were fully defensive in our programming. We didn't explicitly check for all types back in `value-add`. We assumed that if something wasn't a constant, then it must be a range, and that didn't leave any room for a new data type. So how do we fix that?

To say this in more detail, here is what went wrong. A limited precision expression is not a constant, so inside the code we drop down to the else clause. This passes the expression on to the conversion procedure, but it just strips off the tag without checking it. It thus applies a selector to an incorrect data type, accidentally converting this expression to a range with no uncertainty. And this causes an incorrect range to be added up.

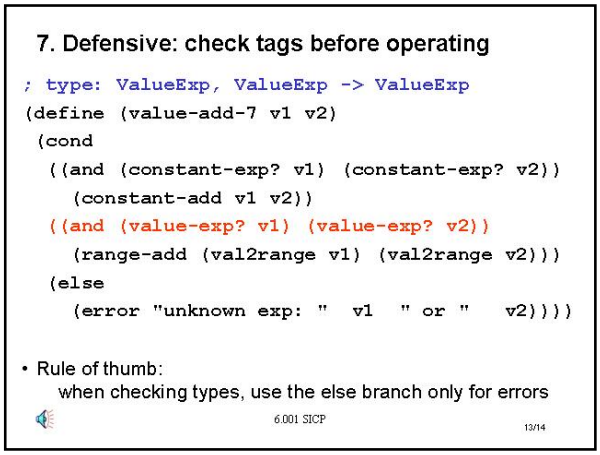
So we need to be fully defensive in all of our procedures!

Slide 9.4.13

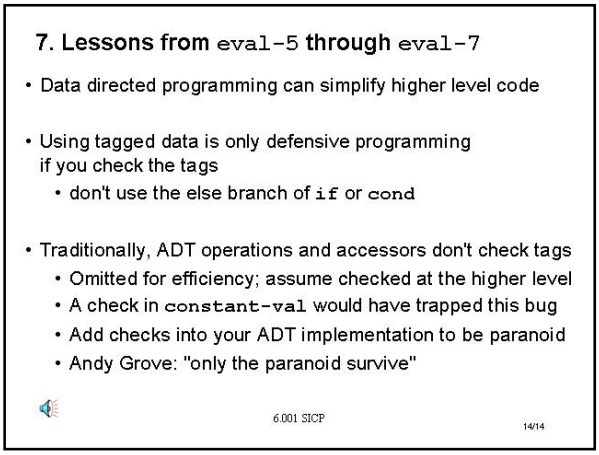
Thus, we should really check all tags before operating on the expressions. And here is associated change to `value-add`.

The main change is to explicitly check if both values are either constants (where we know what to do) or are values. In the latter case we are safe in converting the values to ranges, and adding appropriately. Otherwise, we should complain that we don't know how to combine the two expression types.

Note the general message here. When checking types, we should reserve the failsafe branch only for dealing with errors or unexpected cases.



6.001 Structure and Interpretation of Computer Programs. Copyright © 2004 by Massachusetts Institute of Technology.



Slide 9.4.14

And here are the messages you should take away from this exercise.

As a summary, notice that many programmers don't follow these rules. They often omit type checks for efficiency reasons, and assume that the types will be checked at a higher level in the code. This can easily lead to bugs, and indeed, we would have trapped our bug much earlier, if we had been careful to check types. In short, though it may cost you a little in execution speed, being paranoid is often a great way to efficiently write code that is much less likely to succumb to bugs.

As the founder of Intel notes: "only the paranoid survive"!