

6.001 Notes: Section 8.1

Slide 8.1.1

In this lecture we are going to introduce a new data type, specifically to deal with **symbols**. This may sound a bit odd, but if you step back, you may realize that everything we have done so far in the course has focused on procedures to manipulate numbers. While we have used names for things, we have treated them as exactly that: names associated with values.

Today we are going to create a specific data type for symbols, and see how having the notion of a symbol as a unit to be manipulated will lead to different kinds of procedures.

To set the stage for this, recall what we have when we deal with data abstractions. We said a data abstraction in essence consisted of a **constructor** (for building instances of the abstraction), **selectors** or **accessors** (for getting out the pieces of an abstraction), a set of **operations** (for manipulating the abstraction, while preserving the barrier between the use of the abstraction and the internal details of its representation), and most importantly, a **contract** (specifying the relationship between the constructor and selectors and their behaviors).

Review: data abstraction

- A data abstraction consists of:
 - constructors
 - selectors
 - operations
 - contract



6.001 SICP

1/24

Review: data abstraction

- A data abstraction consists of:
 - constructors


```
(define make-point
  (lambda (x y) (list x y)))
```
 - selectors
 - operations
 - contract



6.001 SICP

2/24

Slide 8.1.2

For example, if I want to create an abstraction for manipulating points in the plane, I could create a constructor like this. `make-point` is a procedure that glues together two parts into a list.

Review: data abstraction

- A data abstraction consists of:
 - constructors


```
(define make-point
  (lambda (x y) (list x y)))
```
 - selectors


```
(define x-coor
  (lambda (pt) (car pt)))
```
 - operations
 - contract



6.001 SICP

3/24

Slide 8.1.3

Here is one of the associated selectors, which in this case takes a data object as built by the constructor, and pulls out the first (or x coordinate) part of that object.

Review: data abstraction

- A data abstraction consists of:
 - **constructors**
`(define make-point
 (lambda (x y) (list x y)))`
 - **selectors**
`(define x-coor
 (lambda (pt) (car pt)))`
 - **operations**
`(define on-y-axis?
 (lambda (pt) (= (x-coor pt) 0)))`
 - **contract**



6.001 SICP

424

Slide 8.1.4

Given that I can build objects of this type, I can define operations on them. Notice that the key point about these things is that they use the selectors to get at the pieces of the data object. For example, in this case we do **not** use `CAR` to get the piece of the object, we use the defined selector.

Slide 8.1.5

... and then the key piece, the contract, the thing that relates the constructor and selectors together. For this example, the contract states that however we glue pieces together using the constructor, applying the first selector to that result will cause the value of the first piece to be returned. So, with these ideas of abstractions in mind, let's turn to introducing a new kind of data structure.

Review: data abstraction

- A data abstraction consists of:
 - **constructors**
`(define make-point
 (lambda (x y) (list x y)))`
 - **selectors**
`(define x-coor
 (lambda (pt) (car pt)))`
 - **operations**
`(define on-y-axis?
 (lambda (pt) (= (x-coor pt) 0)))`
 - **contract**
`(x-coor (make-point <x> <y>)) = <x>`



6.001 SICP

524

Symbols?

- Say your favorite color



6.001 SICP

624

Slide 8.1.6

Let's first motivate why we need a new data type. Suppose I ask you the following question. Think for a second about how you might respond. I, personally, would probably respond by saying **"blue"**.

Slide 8.1.7

Now, what about this question? If you are thinking carefully about this, you ought to respond by saying **"your favorite color"**. So, we say two different things in response to these two questions. What's the difference in the questions?

Symbols?

- Say your favorite color
- Say "your favorite color"



6.001 SICP

724

Symbols?

- Say your favorite color
- Say "your favorite color"
- What is the difference?
 - In one case, we want the meaning associated with the expression
 - In the other case, we want the actual words (or symbols) of the expression



6.001 SICP

8/24

Slide 8.1.8

If you think carefully about it, you should see that in the first case, I got the **meaning** associated with the expression "your favorite color", much like getting the value associated with a name. In the second case, I got the actual expression. The "double quotation marks" in the second case indicated that I wanted the actual expression, while in the first case I want the value associated with it (i.e. the actual favorite color vs. the phrase "favorite color"). So in many cases we may want to be able to make exactly this distinction, between the value associated with an expression, and the actual symbol or expression itself. This is going to lead us to introduce a new data type.

Slide 8.1.9

Now, the question is how do I create symbols as data objects? Well, we already saw one way of doing this, when we defined a name for a value. And we saw that if we wanted to get back the value associated with that symbol (or name) we could just reference it, and the evaluator would return the associated value. But suppose I want to reference the symbol itself. How do I do that? In other words, how do I distinguish between "your favorite color" and "blue" as the value of "your favorite color".

Creating and Referencing Symbols

- How do I create a symbol?
`(define alpha 27)`
- How do I reference a symbol's value?
`Alpha`
`;Value: 27`
- How do I reference the symbol itself?
`???`



6.001 SICP

9/24

Quote

- Need a way of telling interpreter: "I want the following object as a data structure, not as an expression to be evaluated"

```
(quote alpha)
;Value: alpha
```



6.001 SICP

10/24

Slide 8.1.10

Basically, we need to back up and think about what the Scheme interpreter is doing. When we type in an expression and ask for it to be evaluated, the **reader** first converts that expression into an internal form, and the **evaluator** then applies its set of rules to determine the value of the expression.

Here, we need a way of telling the reader and evaluator that we don't want to get the value. Scheme provides this for us with a special form, called `quote`. If we evaluate the example expression using this special form, it returns for us a value of the type **symbol** that somehow captures that name.

Note that it makes sense for `quote` to be a special form. We

can't use normal evaluation rules because that would cause us to get the value associated with the name **alpha** but in fact our goal is to simply keep the name, not its value.

Slide 8.1.11

So what kind of object is a symbol? We can think of it as a primitive data object. Hence it doesn't really have a constructor or selectors, though `quote` serves to help us distinguish between the symbol and its value.

It does, however, have some operations. In particular, the predicate `symbol?` takes in an object of any type, and returns `true` if that object is a symbol.

The operation `eq?` is used to compare two symbols (among other things) and we will return to that in a second.

So here is our new data type for creating symbols, that is, data objects that refer to the name itself, rather than the value with which it is associated.

Symbol: a primitive type

- constructors:
None since really a primitive not an object with parts
- selectors
None
- operations:

```

symbol?      ; type: anytype -> boolean
(symbol? (quote alpha)) ==> #t

eq?          ; discuss in a minute

```



6.001 SICP

11/24

Symbol: printed representation



6.001 SICP

12/24

Slide 8.1.12

To see how this data structure is handled, let's go back to our "two worlds" view of evaluation, separating the visible world of the user from the internal execution world of computation. What happens when we consider symbols in this context?

Slide 8.1.13

First, remember what happened when we evaluated other expressions. For example, if the expression were a `lambda` expression, then the evaluator checked the type of this expression, realized it was a special form, a lambda, and used the rule for that particular special form. In this case, it would create the compound procedure represented by that expression, and return a pointer to that created object, causing the computer to print out information identifying that pointer, i.e. some value associated with such an object.

Symbol: printed representation

```
(lambda (x) (* x x))
#[compound-...]
```

Diagram illustrating the evaluation and printing process for a lambda expression:

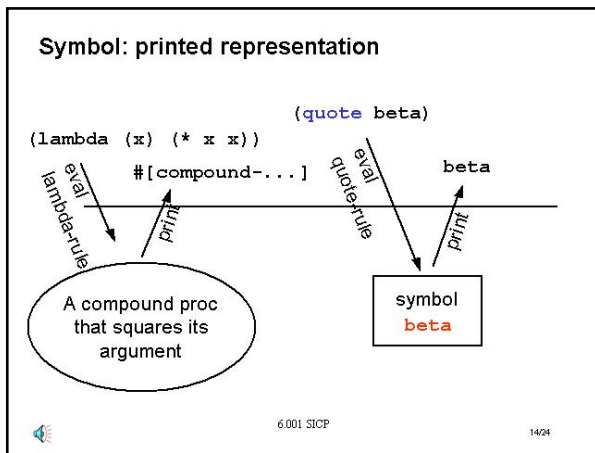
- The expression `(lambda (x) (* x x))` is evaluated using the `eval` function and the `lambda-rule`.
- The result is a compound procedure object, represented by the pointer `#[compound-...]`.
- The `print` function is used to display the printed representation of this object.

A compound proc that squares its argument



6.001 SICP

13/24

**Slide 8.1.14**

Something different happens with **quotes**. If we type in an expression involving the special name `quote`, the evaluator checks the type of this expression, recognizes the special form, and uses a rule designed for such special expressions. In the case of **quote**, we simply take the second subexpression and create an internal representation for it. The reader recognizes this as a sequence of characters and creates a symbol with that sequence of characters, like a name. The evaluator then returns to the visible world something to print out, simply the name that we just quoted, `beta` in this case.

Slide 8.1.15

Now that we have the ability to create this new kind of data object, note that we can use it anywhere we would expect to use such primitive objects. For example, we can certainly create a list of normal things, like numbers. Remember that creating the list of 1 and 2 returns a printed representation of that list structure, written as `(1 2)`.

Symbols are ordinary values

```
(list 1 2)      ==> (1 2)
```

Symbols are ordinary values

```
(list 1 2)      ==> (1 2)
(list (quote delta) (quote gamma))
  ==> (delta gamma)
```

Slide 8.1.16

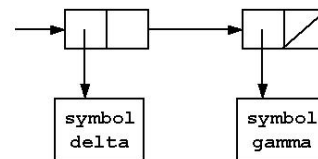
... but I could also create a list of quoted things. We evaluate the arguments to `list`, getting two symbols, then create the list of those symbols, finishing with a printed representation of the structure created by gluing those symbols together.

Slide 8.1.17

What does that list look like? Well, `list` creates a box-and-pointer structure just as in the case of numbers. Thus at the top level of that structure, we will have a skeleton containing two things, ending in the special "empty list" symbol. And what hangs off of this spine? A pointer to the data structure of a symbol! Thus we can use symbols in the same places we might have earlier used numbers within other data structures.

Symbols are ordinary values

```
(list 1 2)      ==> (1 2)
(list (quote delta) (quote gamma))
  ==> (delta gamma)
```



A useful property of the quote special form

```
(list (quote delta) (quote delta))
```



6.001 SICP

19/24

Slide 8.1.18

In fact, our Scheme evaluator is smart, and it keeps track of what symbols have been created so far. As a consequence, when we refer to a symbol, Scheme gives us a pointer to the unique instance of that symbol. We can illustrate that as shown, by evaluating this expression.

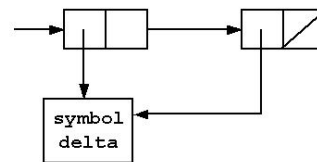
This will create a list of two elements, both of which happen to be the symbol **delta**.

Slide 8.1.19

Scheme will create a box-and-pointer structure for a two-element list. But the `car` of both cons pairs in this list now point to exactly the same object inside the machine, namely the data structure for the symbol `delta`.

A useful property of the quote special form

```
(list (quote delta) (quote delta))
```

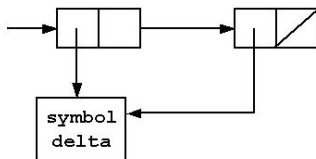


6.001 SICP

19/24

A useful property of the quote special form

```
(list (quote delta) (quote delta))
```



Two quote expressions with the same name return the same object



6.001 SICP

20/24

Slide 8.1.20

This is valuable, because it gives us a way of creating predicates for testing equality of symbols, and indeed other more complicated objects, as we will see a bit later on.

Slide 8.1.21

Our predicate for testing equality of symbols is `eq?`. This is a very powerful procedure, used to test equality of a range of structures, as we will see. `Eq?` is a primitive procedure (i.e. something built into Scheme), and it returns the Boolean value "true" if its two arguments are the same object. For our context, that says that since we create only one instance of each symbol, using `eq?` to test equality of symbols will return true if the two expressions evaluate to a pointer to the same symbol data structure.

The operation `eq?` tests for the same object

- a primitive procedure
- returns `#t` if its two arguments are the same object
- very fast



6.001 SICP

21/24

The operation `eq?` tests for the same object

- a primitive procedure
- returns `#t` if its two arguments are the same object
- very fast

```
(eq? (quote eps) (quote eps)) ==> #t
(eq? (quote delta) (quote eps)) ==>
```



6.001 SICP

22/24

Slide 8.1.22

Here is an example of what we mean by that. If we apply `eq?` to two arguments that evaluate to the same symbol, we get a "true" value returned, otherwise a "false" value is returned.

Slide 8.1.23

Finally, here is the type of `eq?`. It accepts two arguments of **any** type other than a number or a string, and returns a Boolean value, based on the rules described above.

The operation `eq?` tests for the same object

- a primitive procedure
- returns `#t` if its two arguments are the same object
- very fast

```
(eq? (quote eps) (quote eps)) ==> #t
(eq? (quote delta) (quote eps)) ==>
```

- For those who are interested:

```
; eq?: EQtype, EQtype ==> boolean
; EQtype = any type except number or string
```



6.001 SICP

23/24

The operation `eq?` tests for the same object

- a primitive procedure
- returns `#t` if its two arguments are the same object
- very fast

```
(eq? (quote eps) (quote eps)) ==> #t
(eq? (quote delta) (quote eps)) ==>
```

- For those who are interested:

```
; eq?: EQtype, EQtype ==> boolean
; EQtype = any type except number or string
```

- One should therefore use `=` for equality of numbers, not `eq?`



6.001 SICP

24/24

Slide 8.1.24

For numbers, we have a separate procedure to test equality, and a different procedure to testing equality of strings. Everything else uses `eq?`.

This now completes our method for creating and dealing with symbols.

6.001 Notes: Section 8.2

Slide 8.2.1

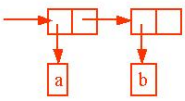
Having the ability to intermix numbers and symbols in expressions is a very useful thing, and as a consequence we would like to be able to generalize this to all sorts of data structures. Since our primary data structure is a list, it would be nice if we had the ability to quote list structure, in addition to simple names.

Generalization: quoting other expressions

6.001 SICP

1/5

Generalization: quoting other expressions

Expression:	Reader converts to:	Prints out as:
1. <code>(quote a)</code>	a	a
2. <code>(quote (a b))</code>		<code>(a b)</code>
3. <code>(quote 1)</code>	1	1

In general, `(quote DATUM)` is converted to `DATUM`



6.001 SICP

2/5

Slide 8.2.2

In fact, our reader and evaluator will do this for us. Since the fundamental representation of expressions in our language is in terms of lists and list structure, the reader is set up to convert every typed in expression into list structure. This is true for any expression created out of parentheses, which denote the boundaries of the list structure.

As we will see in a few lectures, the evaluator is then set up to take that list structure and manipulate it according to the rules of evaluation, to determine the meaning of the expression.

In the case of the special form `quote`, however, the evaluator simply passes on the list structure, without any evaluation. Thus in general, quoting a printed representation of a list structure,

including sublists of numbers and symbols, gets converted to the appropriate list structure internally, and then returned. Its printed representation will then match the original expression.

Slide 8.2.3

This is nice, because `quote` now let's us distinguish between names of things and their values, for virtually every kind of structure. Of course, writing out long expressions involving the special symbol `quote` is a bit tedious, so we have a nice shorthand in Scheme, namely the single quote mark `'`. Thus, `'a` is just a shorthand for `(quote a)`. And `'(1 2)` is just a shorthand notation for `(quote (1 2))` which we already saw is shorthand for `(list (quote 1) (quote 2))`. This means in general that placing a `'` in front of the printed representation for any list structure will cause the evaluator to create the corresponding list structure.

Shorthand: the single quote mark

`'a` is shorthand for `(quote a)`
`'(1 2)` is shorthand for `(quote (1 2))`



6.001 SICP

3/5

Your turn: what does evaluating these print out?

```
(define x 20)

(+ x 3)           ==>

'(+ x 3)         ==>

(list (quote +) x '3) ==>

(list '+ x 3)    ==>

(list + x 3)     ==>
```



6.001 SICP

46

Slide 8.2.4

So let's take a quick break to see if you are getting this idea. Here are a set of expressions. What gets printed out as a result of evaluating each of these? When you think you have the answers, go to the next slide.

Slide 8.2.5

So here are the solutions. First, notice that we have defined x to have the value 20, creating a pairing of that value with that name. Evaluating the first expression just gives us a normal combination, resulting in the addition of 3 to the value of x , or 23. The next expression, with the single quote, says to just return a list whose printed representation is equivalent to this, i.e. the list of '+', 'x' and '3', or the list of the symbol +, the symbol x and the number 3. Thus what is printed out is the same expression as what was quoted.

The next expression draws a distinction with this example. It says to create a list of a quoted +, the **value** of x , and the quoted value of 3. Thus we get a list of the symbol + (because we quoted it), the number 20, since we asked for the value of x , and the number 3.

The next expression returns the same value, since quoting a number just returns the number.

Finally, what happens if we just ask for the list of +, x and 3? Well, we get a list of the values of each of these expressions, as shown.

Thus, these examples show the variations in the use of quotation within list structure, determining when the values of expressions are returned and when the names are simply returned.

Your turn: what does evaluating these print out?

```
(define x 20)

(+ x 3)           ==> 23

'(+ x 3)         ==> (+ x 3)

(list (quote +) x '3) ==> (+ 20 3)

(list '+ x 3)    ==> (+ 20 3)

(list + x 3)     ==> ([procedure #...] 20 3)
```



6.001 SICP

56

6.001 Notes: Section 8.3

Slide 8.3.1

Let's take the idea of symbols, and combine it with some of the other lessons we've seen so far, to see how symbols add to the expressive power of our language. To do this, we'll look at the example of symbolic derivatives, in particular, creating a system to compute symbolic derivatives. By that, I mean returning symbolic expressions, much as you do in calculus. Thus, I want a system that takes some representation for the algebraic expression $x + 3$ and some representation for the variable x , and returns the derivative of this expression with respect to that variable.

Symbolic differentiation

Algebraic expression	Representation
$x + 3$	(+ x 3)
x	x
$5y$	(* 5 y)
$x + y + 3$	(+ x (+ y 3))



1/16

To do this, I am going to need a way of representing expressions, which I will do using lists. Thus the algebraic expression $x + 3$ I choose to represent as the list $(+ x 3)$, that is a list of the symbol, $+$, the symbol, x , and the number 3. For base cases, I will just represent a variable by its symbol. Products I will represent in a similar fashion. And given an expression involving more than two terms I will break into recursive pieces each of which involves at most two terms.

Thus, I am going to restrict my system to sums and products of at most two terms. I haven't said how to build the system, of course, but only how I am going to represent expressions in my system.

Symbolic differentiation

```
(deriv <expr> <with-respect-to-var>) ==> <new-expr>
```

Algebraic expression	Representation
$x + 3$	$(+ x 3)$
x	x
$5y$	$(* 5 y)$
$x + y + 3$	$(+ x (+ y 3))$



2/16

Slide 8.3.2

As we have already said, we would like to build a procedure `deriv` that takes as input some representation of an algebraic expression, and a representation of the variable with respect to which we want to take the derivative, and returns a representation of the new expression that represents the derivative of that algebraic expression.

Slide 8.3.3

So, for example, here is the behavior I would like. I would like to differentiate $x + 3$ with respect to x and get back the value 1. Notice the use of the single quote to indicate that I want the list structure itself as the value of the argument, creating a representation of the algebraic expression. Thus, we want our system to take a symbolic algebraic expression as input, and return a new symbolic algebraic expression as output, satisfying the rules of calculus.

Symbolic differentiation

```
(deriv <expr> <with-respect-to-var>) ==> <new-expr>
```

Algebraic expression	Representation
$x + 3$	$(+ x 3)$
x	x
$5y$	$(* 5 y)$
$x + y + 3$	$(+ x (+ y 3))$

```
(deriv '(+ x 3) 'x) ==> 1
(deriv '(+ (* x y) 4) 'x) ==> y
(deriv '(* x x) 'x) ==> (+ x x)
```



3/16

Building a system for differentiation

Example of:

- Lists of lists
- How to use the symbol type
- symbolic manipulation

1. how to get started
2. a direct implementation
3. a better implementation



4/16

Slide 8.3.4

To build this system, I am going to stitch together several ideas, using lists of lists to represent expressions, using symbols to capture algebraic expressions, and using procedural abstractions to manipulate the list structures corresponding to those expressions.

To build the system, I am going to consider several stages, focusing on how to initially get things going, then how to build a direct implementation, and finally how to learn from the direct method to create a better implementation. Throughout, we will see how these ideas of data structures and procedural abstractions work together to implement this system.

Slide 8.3.5

First, I need to figure out exactly what I want in terms of the behavior of my system. So let's look carefully at what the pieces are. For primitive **algebraic** expressions, we know that there are some simple rules. The derivative of a **constant** with respect to any variable is just zero. The derivative of a variable with respect to itself is just one, while the derivative of any other variable with respect to this variable is also zero. Note that we are inherently assuming that a variable is not a function of some other variable.

For more complicated expressions, we have some very nice rules. For example, to get the derivative of a sum with respect to some variable, we know that we can take the derivative of the two parts of the sum, then add those results back together to get the final expression. For the derivative of a product, a slightly more complicated rule states that we can take the derivatives of the pieces of the product, multiply the results by appropriate other parts, then add the result together to get the final value. Note that this should look familiar. Notice what we are doing. We are taking a complex thing, reducing it to simpler versions of the same operation on smaller pieces, and then gluing the parts back together again. So to apply derivatives to complex things, we do this recursively on simpler pieces.

Here then are some rules for the overall behavior I want my differentiation system to obey.

1. How to get started

- Analyze the problem precisely

deriv constant dx = 0
 deriv variable dx = 1 if variable is the same as x
 = 0 otherwise

deriv (e1+e2) dx = deriv e1 dx + deriv e2 dx
 deriv (e1*e2) dx = e1 * (deriv e2 dx) + e2 * (deriv e1 dx)



5/16

1. How to get started

- Analyze the problem precisely

deriv constant dx = 0
 deriv variable dx = 1 if variable is the same as x
 = 0 otherwise

deriv (e1+e2) dx = deriv e1 dx + deriv e2 dx
 deriv (e1*e2) dx = e1 * (deriv e2 dx) + e2 * (deriv e1 dx)

•Observe:

- e1 and e2 might be complex subexpressions
- derivative of (e1+e2) formed from deriv e1 and deriv e2
- a [tree problem](#)



6/16

Slide 8.3.6

We can observe several useful things about what we have done. First, note that e1 and e2 might themselves be complex expressions, in which case we would apply these rules again to each of those pieces. Thus, our procedure will need to recursively walk down these expressions, applying these rules to subsequent pieces.

Second, as we noted, the derivative of a sum is decomposed into simpler versions of the same problem on smaller pieces, plus a simpler operation that puts the results back together. Putting these two observations together, we can see that expressions might not be lists, but lists of lists, sometimes

called **trees**, of arbitrary depth. That is, we can apply these rules to expressions whose parts are themselves sums or products of elements who might be sums or products, and so on. We simply want to recursively apply the rules to break the problem down into simpler pieces until we ultimately reach primitive cases, then glue all the parts back together again.

Slide 8.3.7

Given our suggestion that we can represent expressions as lists of things, that is lists of subexpressions with the symbol for the operator at the front of the list, and the arguments behind it, we can nicely associate a data type with each expression.

First, any symbol will represent a variable. Any number will simply denote a constant. And then any other expression denotes its type by the object at the front of the list. Thus, any expression beginning with a + is a sum, and thus will be subject to the rule for derivatives of sums. Similarly for products. And, of course, the subexpressions could themselves be lists whose type is indicated by the type of the first part of the list. In other words, except for our primitive expressions (constants and

variables), every expression in our system has its type defined by the first subexpression of the list. This also excludes some kinds of expressions that might be perfectly valid from an algebraic perspective. Thus, expressions with the operator in the middle are not included in our choice of representation. Also, we have restricted ourselves for convenience to expressions with exactly two arguments. Algebraic expressions with more than two parts will have to be represented by nested lists of operations.

Note that we are simply making some design choices in our system, something we are free to do as we set up the computational structure for implementing that system. Thus our choice is to allow as legal expressions, constants, variables or lists of three elements, the first of which is either a + or a *, and the other two of which are legal expressions, by this definition.

Type of the data will guide implementation

- legal expressions

x	(+ x y)	
2	(* 2 x)	(+ (* x y) 3)
- illegal expressions

*	(3 5 +)	(+ x y z)
()	(3)	(* x)

7/16

Type of the data will guide implementation

- legal expressions

x	(+ x y)	
2	(* 2 x)	(+ (* x y) 3)
- illegal expressions

*	(3 5 +)	(+ x y z)
()	(3)	(* x)

```

; Expr = SimpleExpr | CompoundExpr
; SimpleExpr = number | symbol
; CompoundExpr = a list of three elements where the first
                  element is either + or *
; = pair< (+|*), pair<Expr, pair<Expr,null>>>

```

9/16

Slide 8.3.8

So let's formalize this with a type description for our legal expressions in this simple little language of derivatives that we are building.

In our system, an expression, which we denote by `Expr` to distinguish them from more general Scheme expressions, can either be a simple expression or a compound one. The | symbol denotes "or", by the way. And what are those? Well a simple expression is either of type number or symbol (corresponding to our constants and variables). A compound expression now has a very particular type. It is a list of three elements, where the first element is either the symbol + or *

and the other two parts are expressions as defined by this type definition. Note the format for specifying that this is a list of three elements, by using our type notation for pairs.

Note how this type definition is recursive, thus automatically allowing for arbitrary depth expressions.

Slide 8.3.9

So now we can put together an initial plan for implementing this system. Since we only have a few kinds of expressions, the simplest approach is to use a different procedure for taking the derivative of each kind of expression. So we could define `deriv` to be a procedure (note the `lambda`) that accepts an expression and a variable (represented using the forms we just discussed), and checks to see if the expression is a simple one. If so, we could have one procedure for handling such expressions, otherwise we could design a second procedure to handle compound expressions. Thus, we have used the fact that there are two general types of expressions to design our procedure. All we have to do is decide what it means for an expression to be simple.

2. A direct implementation

- Overall plan: one branch for each subpart of the type

```
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      <handle simple expression>
      <handle compound expression>
      )))
```



9/16

2. A direct implementation

- Overall plan: one branch for each subpart of the type

```
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      <handle simple expression>
      <handle compound expression>
      )))
```

- To implement `simple-expr?` look at the type
 - CompoundExpr is a pair
 - nothing inside SimpleExpr is a pair
 - therefore

```
(define simple-expr? (lambda (e)
  (not (pair? e))))
```



10/16

Slide 8.3.10

For that, we can just look at the type. What do we know about these expressions? Well, we know that a compound expression is a list, hence starts with a pair, and none of our simple expressions is a pair. So to find a simple expression, we could simply confirm that the expression is not a pair.

Slide 8.3.11

Now we can start completing the implementation by filling in the cases. The first branch of our cases deals with simple expressions. And here we can simply set up a branch to deal with each specific type. Since there are only two types of simple expressions, we can just check to see if we are dealing with number, say, in which case we will apply the appropriate rule, otherwise we will apply the rule for variables. And how do we handle each simple case? We simply go back to our rules from our problem design and description.

Simple expressions

- One branch for each subpart of the type

```
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      (if (number? expr)
          <handle number>
          <handle symbol>
          )
      <handle compound expression>
      )))
```

- Implement each branch by looking at the math



11/16

Simple expressions

- One branch for each subpart of the type

```
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      (if (number? expr)
          <handle number> 0
          <handle symbol> (if (eq? expr var)
                              1 0))
      <handle compound expression>
      )))
```

- Implement each branch by looking at the math



12/76

Slide 8.3.12

We said the derivative of a constant (or number) was just 0, so we can simply fill in that case.

We said the derivative of a variable was 1, if it was the same variable as that with respect to which we are differentiating, otherwise it was 0. To fill in that case, we just need to test if the expression is the same variable as the supplied variable, and for that we use `eq?` since our variables are represented as symbols. And that is all we need to handle simple expressions!

Slide 8.3.13

For compound expressions, we can use exactly the same design methodology. We can have a different branch of this top level decision procedure for each type of expression. Since we only have two types of compound expressions, we could simply check to see if the expression is a **sum** or not. To see if the compound expression is a sum, we will simply grab the first subexpression (using `CAR` since we know it is a list) and test to see if it is the symbol `+` by comparing using `eq?`. Notice the use of the `'` mark to give us the symbol `+` for the comparison. Based on that decision, we will either handle the expression as a sum or as a product.

Compound expressions

- One branch for each subpart of the type

```
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      (if (number? expr) 0
          (if (eq? expr var) 1 0))
      (if (eq? (car expr) '+)
          <handle add expression>
          <handle product expression>
          )))
```



13/76

Sum expressions

- To implement the sum branch, look at the math

```
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      (if (number? expr) 0
          (if (eq? expr var) 1 0))
      (if (eq? (car expr) '+)
          (list '+
                (deriv (cadr expr) var)
                (deriv (caddr expr) var))
          <handle product expression>
          )))
```



14/76

Slide 8.3.14

We can keep working our way through the implementation. For example, to deal with "sum" expressions, we can go back to what our formal math analysis said. In particular, we need to take the derivatives of the subexpressions, then "add" them together (symbolically). To do this, we take the `cadr` of the expression, which gets out the first part of the sum, and apply `deriv` to it to get the symbolic derivative. We do the same thing with the other part of the sum. If we implement `deriv` correctly, this should recursively return symbolic expressions for each part. Then, to create the symbolic sum, we need to convert the result to the appropriate form, namely a list with the

symbol `+` at the front. Notice how `deriv` thus decomposes the problem into simpler versions of the same problem, and then constructs a new form to return, based on those parts.

Slide 8.3.15

So now let's try it out, using the example of the derivative of $(+ x y)$ with respect to x . Instead of getting what we might expect mathematically, namely 1 , we get $(+ 1 0)$.

Technically these are the same thing, but the returned form is not as satisfying as just returning the simplest possible form of this expression. Notice why this happens. Our procedure always blindly breaks out the pieces of a sum, applies `deriv`, and then glues back together. It doesn't try to simplify the result. The underlying reason, which often happens in direct implementations of methods, is that the list structure of the input expression will be exactly preserved in the output, we simply replace the expression at each element of the list with that expression's derivative.

What if instead we wanted our system to simplify things to more basic terms, thus not preserving the list structure of the input expression?

Sum expressions

- To implement the sum branch, look at the math

```
(define deriv (lambda (expr var)
  (if (simple-expr? expr)
      (if (number? expr) 0
          (if (eq? expr var) 1 0))
      (if (eq? (car expr) '+)
          (list '+
                (deriv (cadr expr) var)
                (deriv (caddr expr) var))
          <handle product expression>
          )))
```

```
(deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)
```

15/76

The direct implementation works, but...

- Programs **always** change after initial design
- Hard to read
- Hard to extend safely to new operators or simple exprs
- Can't change representation of expressions
- Source of the problems:
 - nested if expressions
 - explicit access to and construction of lists
 - few useful names within the function to guide reader



16/76

Slide 8.3.16

We'll consider that question in the next section. But first, let's pull out the key lessons we have seen in taking a direct approach to implementing a system.

First, in almost any system, our program will change after our initial design. In this case, we made an assumption that we didn't realize, namely that the structure of the input list would be preserved. We didn't observe this till we ran some test cases (which is often the case in real systems), and now we need to try to go back and change our code to reflect a better design. But this is hard in this case, mostly because our code as it stands is hard to read, that is, to figure out which parts of the code are handling which parts of the problem. Moreover,

suppose we want to add new expressions to our system, things other than sums and products. This is hard to do, because we have built our code explicitly on the assumption that there are two choices for expressions. And suppose that we decide we want to change the representation of our expressions, for example, to put the operator in the middle, more like real algebraic expressions. This is hard to do because we have used the actual list selectors and constructors directly, rather than isolating them behind a data abstraction.

So the bottom of the slide lists a summary of the causes of these problems, that is, the things we should probably change to build a more flexible system.

6.001 Notes: Section 8.4

Slide 8.4.1

So let's take another run at trying to build a symbolic differentiation system. Let's create a new implementation that takes advantage of these lessons. In particular, we need a better top-level design decision based on types of expressions. We'll use `cond` to handle our decisions based on types, giving us more flexibility than having just two choices at each stage, which is what `if` assumes. And, we'll isolate our data representation from its use, by building a true data abstraction, with an abstraction barrier between the user and the implementer.

3. A better implementation

1. Use `cond` instead of nested `if` expressions
2. Use data abstraction



17

3. A better implementation

1. Use `cond` instead of nested `if` expressions
2. Use data abstraction

•To use `cond`:

- write a predicate that collects all tests to get to a branch:

```
(define sum-expr? (lambda (e)
  (and (pair? e) (eq? (car e) '+))))
; type: Expr -> boolean
```



27

Slide 8.4.2

If we are going to use `cond` to handle the dispatch to methods for different types, we'd like to have predicates that identify explicit types. So we'll gather together into a single procedure all the tests we need to identify a specific type of expression, for example, to determine if an expression is a "sum".

And we'll do the same thing for every other type of expression we want to handle.

Slide 8.4.3

While it is easy to see that we need to do this for compound expressions, note that we have an implicit assumption in our earlier implementation about the representation of simple expressions. For example, we have directly relied on the fact that a variable was represented as a symbol. But we should really isolate this fact and check explicitly for types of simple expressions as well.

Thus to check if an expression is a variable, we should actually first check that it is not a compound expression, then check that it is actually of the form we are using to represent variables, in this case, symbols. Note the use of the predicate `symbol?` to do this.

3. A better implementation

1. Use `cond` instead of nested `if` expressions
2. Use data abstraction

•To use `cond`:

- write a predicate that collects all tests to get to a branch:

```
(define sum-expr? (lambda (e)
  (and (pair? e) (eq? (car e) '+))))
; type: Expr -> boolean
```

- do this for every branch:

```
(define variable? (lambda (e)
  (and (not (pair? e)) (symbol? e))))
```



37

Use data abstractions

- To eliminate dependence on the representation:

```
(define make-sum (lambda (e1 e2)
  (list '+ e1 e2))

(define addend (lambda (sum) (cadr sum)))
```



47

Slide 8.4.4

The second thing we need to do is truly implement a data abstraction. We need to eliminate the dependencies within the code on the explicit form of the representation. Thus, within the code we should only be using constructors and selectors, which will shield the choice of representation from the use of the abstraction.

Here, for example, is a constructor for "sum" expressions, with one of the associated selectors. By creating this barrier between code that uses expressions (e.g. `deriv`) and the actual representation, we are now free to change that representation. As long as the contract between constructors and selectors is preserved, the code that uses the abstraction will still work.

Obviously we could complete this representation for sums, for products, and for any other expressions we want in

our system.

Slide 8.4.5

Now let's pull these new ideas together into a better `deriv` procedure. The arguments are the same as before. The top level structure, however, has a different form. Here we have a large `cond` expression, where each clause dispatches on a different type of expression. Notice the nice form here. Each clause has a predicate that checks the type for each kind of expression, starting with the simpler expressions. Associated with each type is the method to apply for that type. Notice in particular the form for "sums". We use the selectors to get out the pieces, we recursively apply `deriv` to those pieces, then we use the constructor to glue the pieces together into the correct form.

So why bother to go to all this trouble? Since by doing this,

`deriv` only uses selectors to get at pieces, we can now freely change the underlying representation without causing any damage to `deriv` or any other procedure that uses the selectors and constructors.

A better implementation

```
(define deriv (lambda (expr var)
  (cond
    ((number? expr) 0)
    ((variable? expr) (if (eq? expr var) 1 0))
    ((sum-expr? expr)
     (make-sum (deriv (addend expr) var)
                (deriv (augend expr) var)))
    ((product-expr? expr)
     <handle product expression>)
    (else
     (error "unknown expression type" expr)))
  ))
```

Slide 8.4.6

Let's drive this point home with an example. Here again is our original example, including the case where it seemed to return the "wrong" answer. Having separated out a clean data abstraction, we can fix this problem very easily, without having to touch `deriv`.

In particular, let's change our constructor. When we go to create a sum, let's first check to see if we can simplify the expression. So instead of just creating a list starting with the symbol "+", we'll first see if the two other expressions are numbers. If they are, let's actually add them together. Look very carefully at this. In the case that both expressions are numbers, we return as the expression the **value** obtained by applying the operator associated with + to

those values, that is, we don't create a list here, we return a numerical value!

In the other cases, we do return a symbolic expression, simply choosing to put the number first if there is one. The key issue is that we have only changed one thing in our system, a constructor. What happens to the full system?

Slide 8.4.7

.. well it nicely gives us the change we wanted. In this simple case, it returns the value of the numeric sum.

To summarize, by isolating data representations from data use, it becomes **much** easier to make changes in the behavior of our system without requiring detailed and intertwined coding changes. This leads to cleaner code, which is much easier to maintain and modify.

Isolating changes to improve performance

```
(deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)
(define make-sum
  (lambda (e1 e2)
    (cond ((number? e1)
           (if (number? e2)
               (+ e1 e2)
               (list '+ e1 e2)))
          ((number? e2)
           (list '+ e2 e1))
          (else (list '+ e1 e2)))))
```

Isolating changes to improve performance

```
(deriv '(+ x y) 'x) ==> (+ 1 0) (a list!)
(define make-sum
  (lambda (e1 e2)
    (cond ((number? e1)
           (if (number? e2)
               (+ e1 e2)
               (list '+ e1 e2)))
          ((number? e2)
           (list '+ e2 e1))
          (else (list '+ e1 e2)))))

(deriv '(+ x y) 'x) ==> 1
```