Classes 1 and 2, January 25 and 30, 2024

We begin by describing the purely functional sublanguage of R5RS Scheme

'Purely functional' means NO ASSIGNMENT and, more generally, no 'destructive programming': after initial definition, values are not changed (or 'destroyed').

We focus on what is called DECLARATIVE PROGRAMMING in contrast to the IMPERATIVE PROGRAMMING style of prev. class

At the basis of this language are a few primitive data types. Initially, we'll make use of just three of these:

- numbers
- booleans
- functions

Numbers can be integers, rationals, or reals (complex, too, but we won't use these)

integers: 'infinite' precision
rationals: 1/3 displays as 1/3, not as 0.33333...
reals: subject to same precision limitations as in C

Booleans are just #t (true) and #f (false) — not themselves defined

This is one difference between R5RS and the version of MIT Scheme used in Abelson & Sussman (SICP). They use nil for #f, as well as for the empty list. Note that nil is not even defined in R5RS.

Functions come in two main flavors — primitives and user-defined. For this introductory look at the basis of R5RS, we'll focus on the arithmetic primitives.

Examples of these are +, *, /, -, modulo, quotient, remainder, ...

I urge you to get in the habit of taking incremental reads of the R5RS manual, where you will find a complete list of the primitives.

Numbers, Booleans and primitive functions can be said to be self-evaluating: type one in, and get it back, unchanged.

We can (of course) use these primitives to form more complex expressions

Before showing some of these, let me emphasize the critical role of parentheses in Scheme. Unlike their usage in math, parens are NOT just for the convenience of the reader.

When we write

```
(+ 2 3)
```

for example the parens are regarded as an instruction:

- evaluate the leftmost entry - throw an error if its value is not a function
- evaluate the remaining entries
- apply the value of the leftmost entry to these values
- return the result

For (+ 2 3), + evals to the built-in + function, 2 evals to the internal representation of 2, and 3 to the internal rep. of 3. The application then returns 5.

Similarly for *, -, /, ...

(- 5 2) returns 3

(/ 10 2) returns 5

and so on.

Interestingly, these primitives are generic, in at least two ways.

First (unlike C), + is happy to accept reals and rationals as well as integers; indeed, even mixtures of these

(□r will record that the lecture moved over to racket for some examples.)

☐ main takeaways from first excursion
→ generic as regards numeric types
(ie, we do not need separate functions
+-int, +-real, +-rat ...)
and allows mixed types
→ despite the absence of
user-created type tags, types
are nonetheless present: one
cannot compute (+ #t #f), for
example — the primitive +
requires that its arguments be
numbers.

dynamic typing — types detected at run time, rather than at compile time

A second kind of genericity has to do with the number of arguments:

☐ main takeaways from second excursion

→ generic also as regards the number of arguments. Eg (+ 1 2 3) computes 6. The idea is that + regards its args as a list, and it returns the sum of the numbers in that list.

On this model, (+) ought also

make sense, and indeed (+) returns 0 — the sum of all the numbers in the empty list.

Similarly for the other arithmetic primitives? Explore!

Note that (*) returns 1, not 0, in line with mathematical convention.

Even more complex expressions can be formed if we nest parens. For example,

(+ (* 3 4) 2)

is eval'd using the same rule

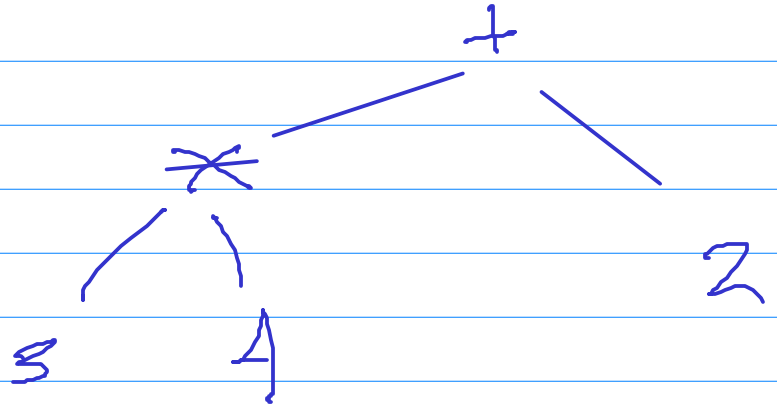
(The Scheme Evaluation Rule)

- + evals to the built-in + function (call it plus)
- 2 evals to two

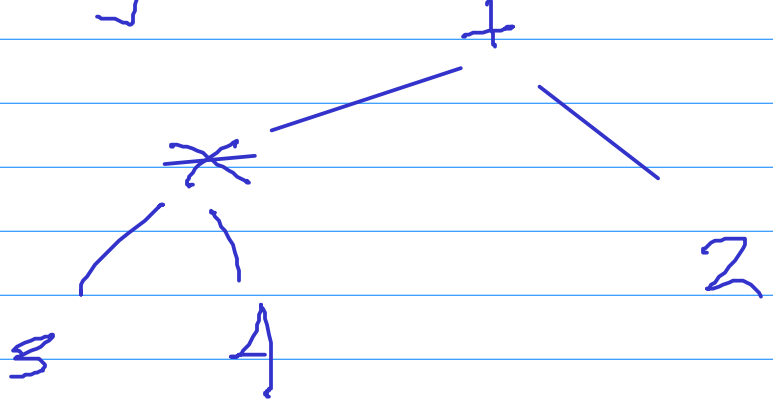
- The rule is applied recursively to get the value 12 for (* 3 4).

So (+ (* 3 4) 2) = 14

There is no limit on the nesting depth, but as these expressions become larger, it will be helpful to have a simple visualization for them. We use trees in the expected way:



It is important to know that one should not expect that + evaluates its args in left to right order. In fact, we know nothing at all about the eval. order



except that the arguments to a function (as well as the function itself) must have been evaluated before the function can be applied.

In fact, this is the ONLY notion of order which matters for this kind of programming. As you will soon see, this explains why the semicolon (;) has in Scheme

(or any functional language) nothing like the importance it has in C. Essentially, 'before and after' are concerns in Scheme only as they apply to the evaluation of a function and its args before the function can be applied to these arguments.

In other words, "P; Q" (do P, then Q) makes NO sense in (pure functional) Scheme!!! ... construed as

We'll have a lot more to say about this.

We will see how function composition makes up for this: assuming

f and g are composable functions,

$(f \cdot g)(x) = f(g(x))$

Next, lets look at variables.

We'll get at variables via the special form define.

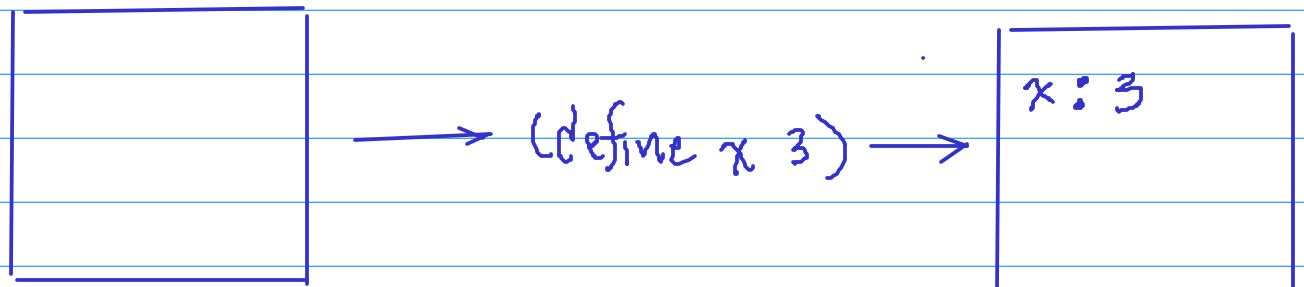
For example

```
(define x 3)
```

We say that define is a special form because the Scheme Evaluation Rule discussed last time does not apply for define.

So - how does define work?

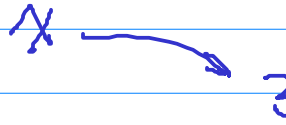
Initially, scheme opens a global namespace in which the primitives (etc) are defined, but no user variables are defined.



initial (empty) environment

We write x : 3 to indicate that x is a name of the value 3

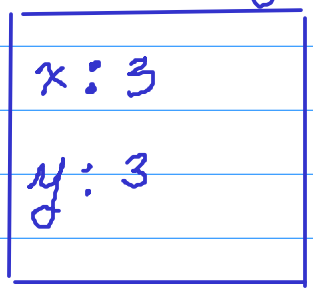
We say "x is bound to 3". Another suggestive picture is



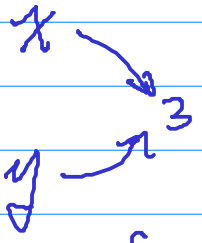
If we were now to

(define y 3)

then the global env would be



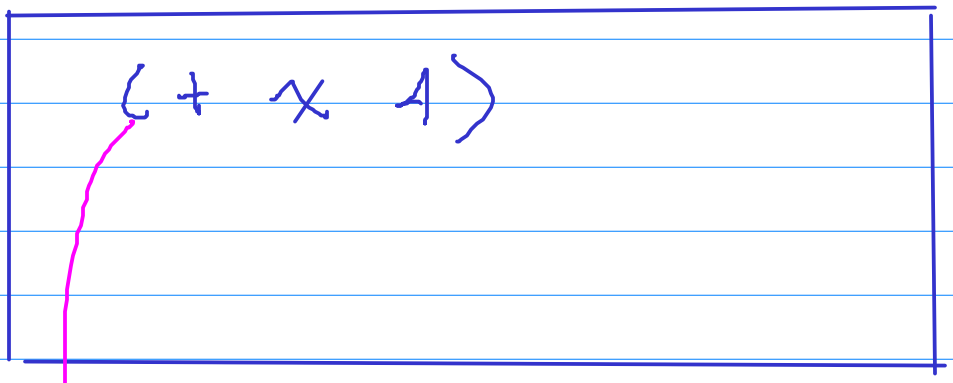
with as the corresponding pointer diagram.



Once (define x 3) has been evaluated, then, within the scope of this define, you may replace all occurrences of x by 3 without changing the meaning of the substituted expression

(define x 3)

scope of this definition



(*)

means (+ 3 4) : anywhere x occurs, we can replace it with a 3. This illustrates REFERENTIAL TRANSPARENCY

So what? Well, consider the contrast with C. After the declaration

int x = 3

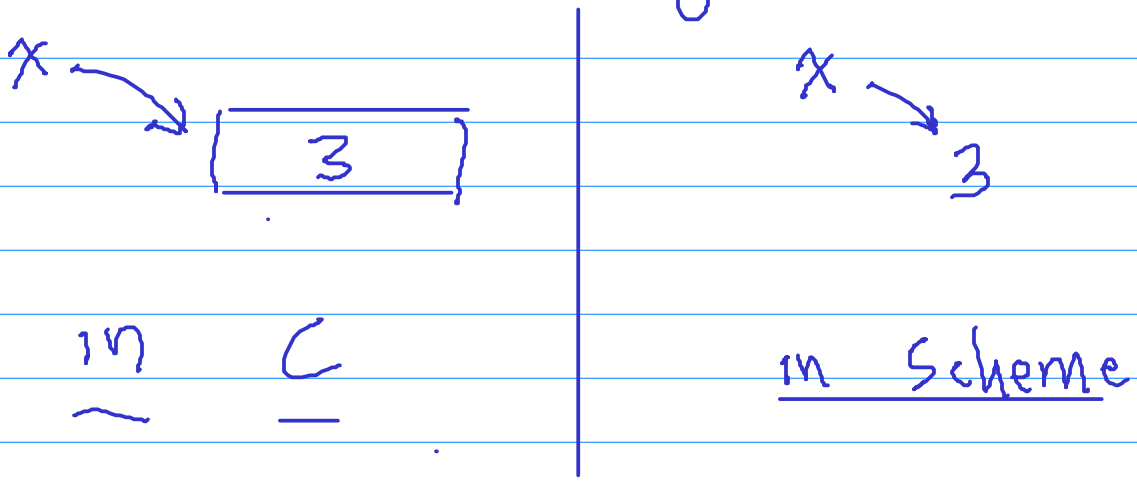
consider

x = x + 1.

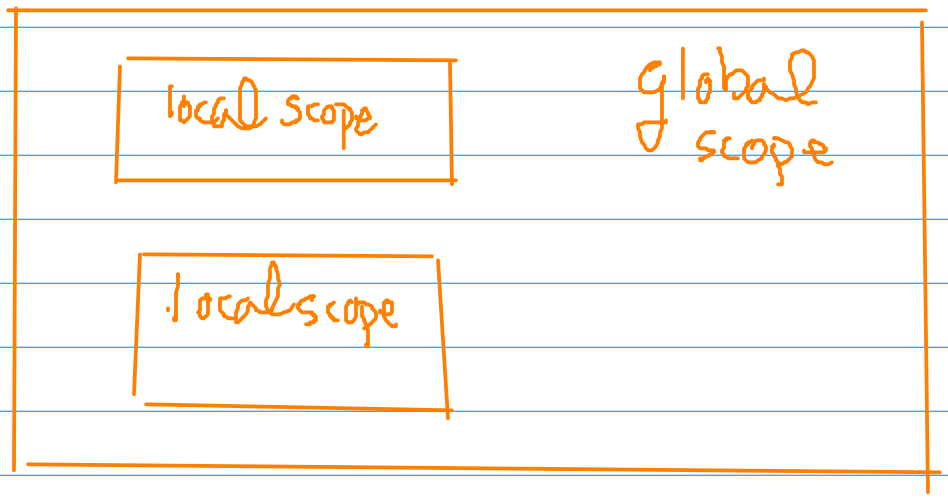
Can we replace x by 3 without wrecking the meaning? Clearly NOT!

What's responsible for the difference?

In C, x is NOT a name for 3 after this initialization — rather it is a name for a box (memory location) containing 3.

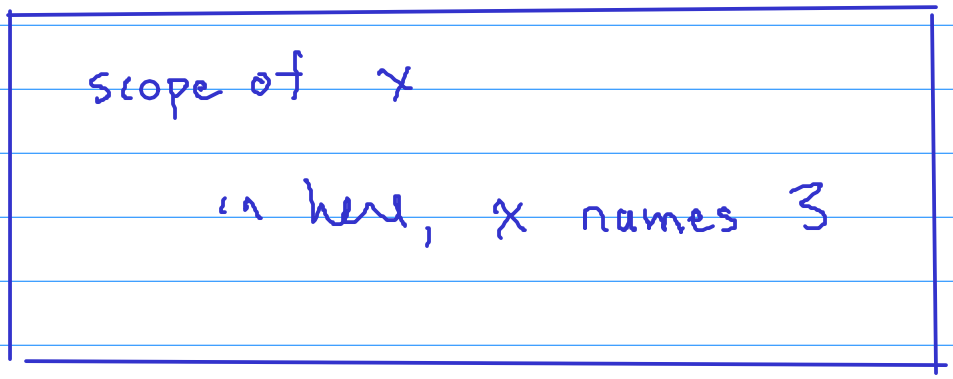


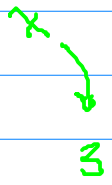
(*) In scheme, as in C, scopes frequently have "holes", typically brought about by local variables



One way if changing x and its scope:

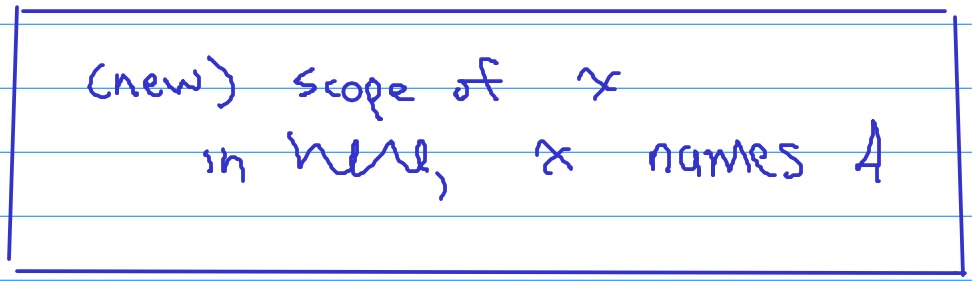
```
(define x 3)
```

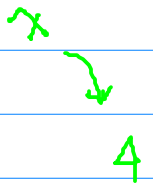





```
(define x 4)
```

previous pointer disappears





Hey! Why isn't this just assignment? Well, it is: it is definitional assignment, making an association between x and its value.

But it is extremely restricted, and Scheme in no way allows anything

equivalent to $x = x + 1$. In particular, define cannot occur within an expression — nor in the body of a program.

What haven't we explained yet about define?

What about

```
(define x (+ x 1))
```

?

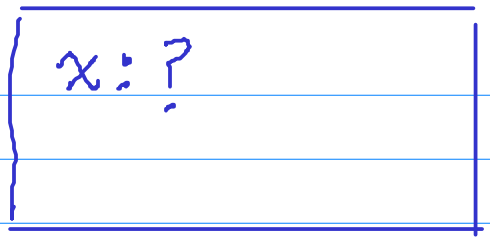
This would assume that x has already been defined: The eval rule for

```
(define x e)
```

(e: some perhaps complicated expression)

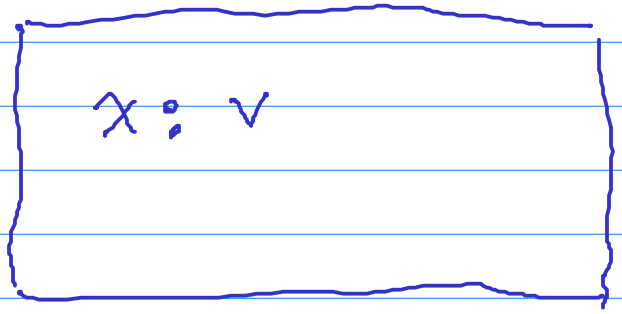
is something like

1. install x in the global env, perhaps without a value



(2) Evaluate e to get its value, say v

(3) Finish the binding - so that, when the eval. of the def is complete, we would have



, i.e., x → v

(Take a moment to think why the standard evaluation rule would not work for define.)

(define x e)

The standard rule would say

- eval the primitive define
- eval x — so x has no value! ← (but x is usually here being defined for the first time)
- eval e

standard Rule breaks here! →

Once a variable has been bound to a value, we can use it anywhere we like as (eg) a shorthand for that value: after

(define pi 3.1415 ---)

we could write (+ 2 pi)

(+ 2 (* 3 pi))

⋮

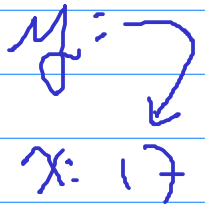
Another example

(define x 17)

Could we then write

(define y x) ?

YES!



① We talked about the special form define – but although I mentioned the severe restrictions on the use of define, I still need to give you an example of forbidden usage of define in an expression

Here's one

```
(+ (define x 3) 2)
```

You can use the Check Syntax button in drracket to see the explanation of why this won't work.

Recall the C distinction between expressions and statements. Pure functional scheme does not have statements.

So this restriction on define is very strong!

Should also have mentioned that 'define' may not be used as a variable - it is a reserved word (**)

(**) Just about any combination of numbers and characters can be used to form identifiers in Scheme. There are some restrictions

→ no embedded white space

→ no embedded parens

→ perhaps others, but I am not thinking of them just now *

By default, Scheme does not differentiate upper and lower case letters.

* You will find the complete set of rules in the R5RS reference manual, which I have posted on Teams.

Another special form is

lambda

When we enter

(lambda (x) (* x x))

in R5RS, the system returns

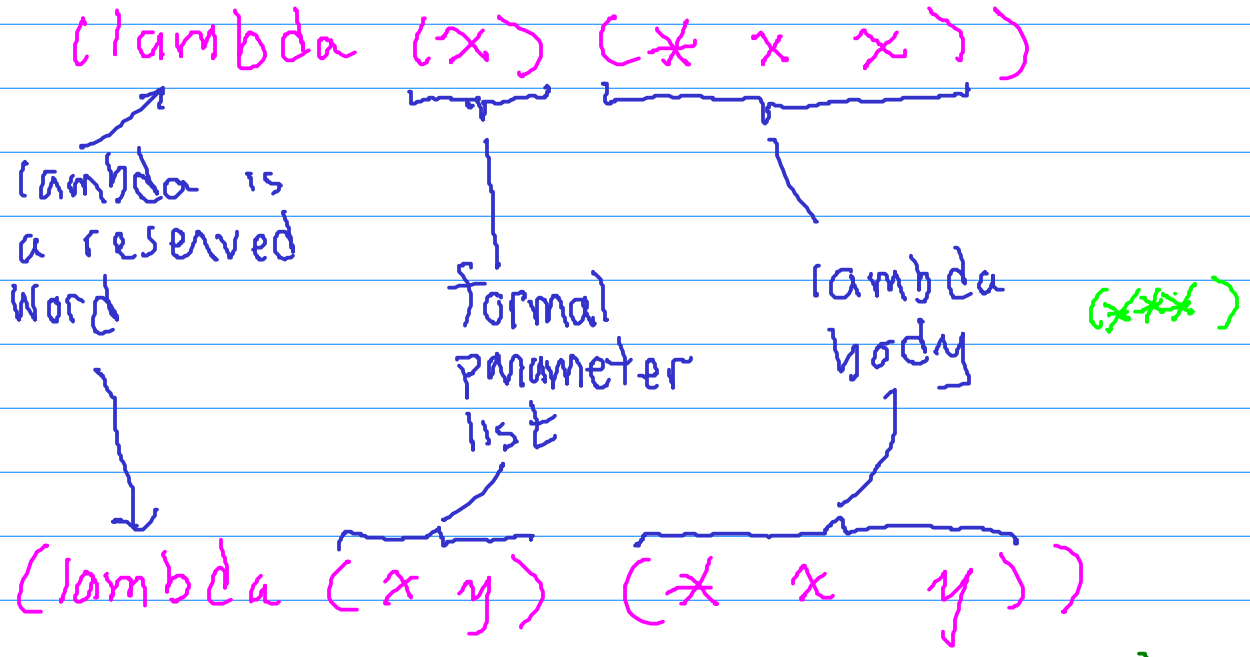
#procedure

We conclude that the lambda form is a 'stand-alone' scheme value.

In fact, such forms are scheme values in the same sense that, eg, numbers and booleans are scheme values. We will see that, just as numbers and booleans can be returned by function calls, so too can functions be returned by function calls. In fact, functions can occur as components of data structures — eg — we will see lists of functions.

One says that, in scheme, functions are first class.

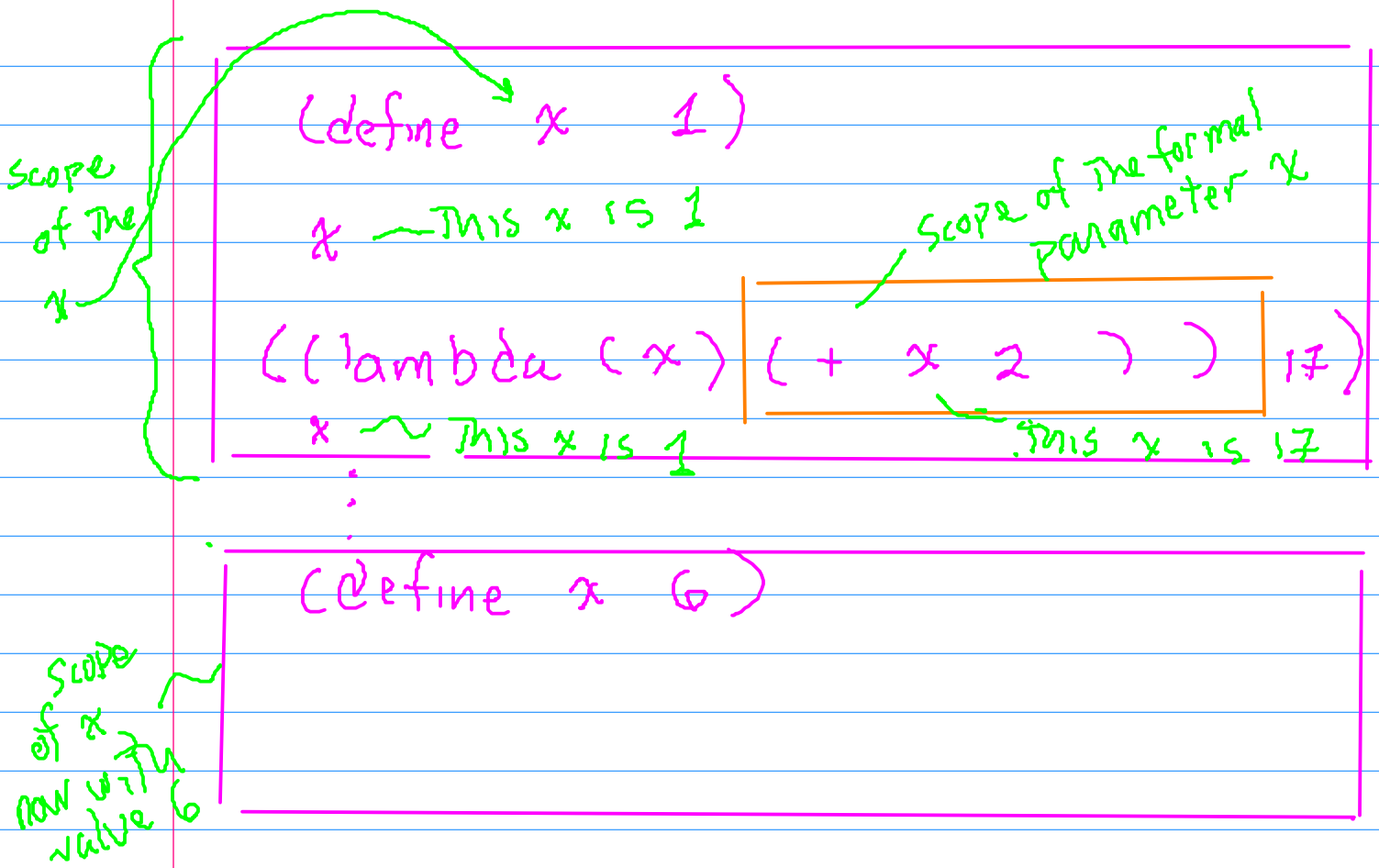
It's useful to take the lambda form apart:



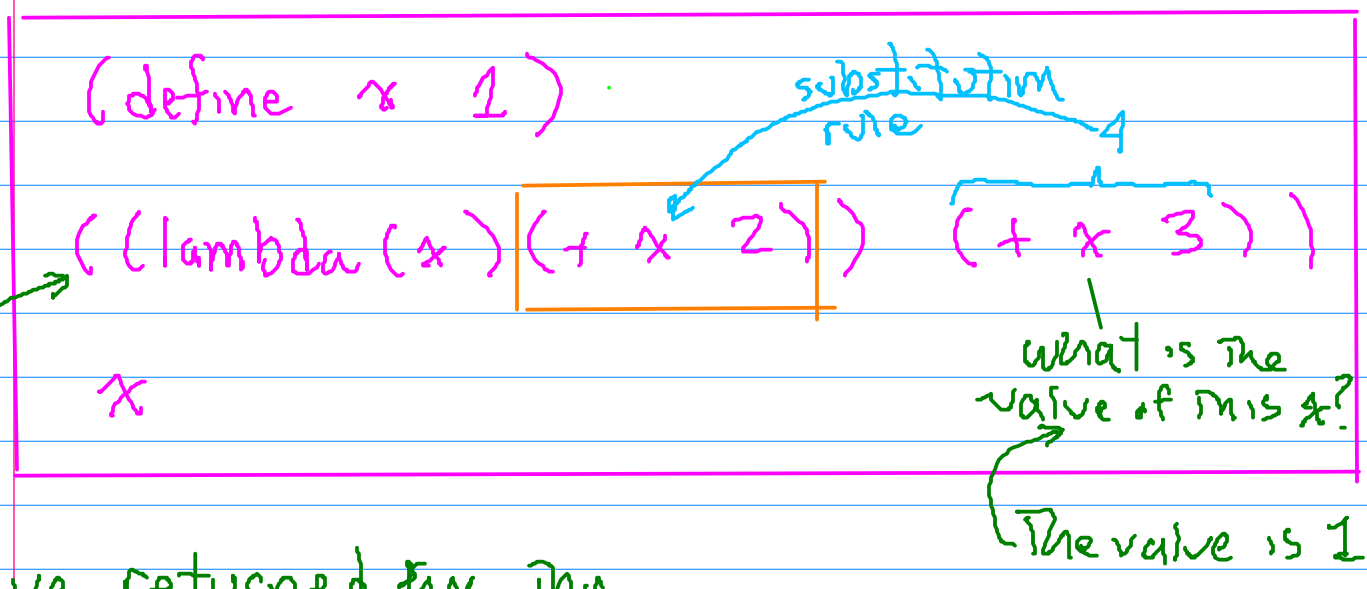
The formal parameters each open a local scope, which is precisely the body of the lambda form.

Returning to the Lecture 1 discussion, these scopes can be said to be holes in any containing scope of the same variable.

For example:



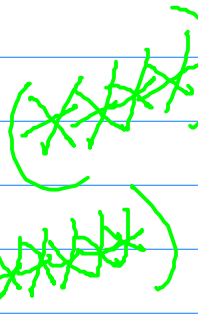
You will want to experiment .. consider



(Anything of the form

```
(fn arg1 arg2 ... argk)
```

is said to be an application.)



Note that define and lambda play nicely together :

```
(define square
  (lambda (x) (* x x)))
```

Abelson & Sussman notation for this function definition is

```
(define (square x)
  (* x x))
```

They prefer to not write the lambda.
(This is the first instance we've seen of what is called <u>syntactic sugar</u>)

Having defined square this way, one uses it as follows

(square 2)

By the scheme eval. rule, square would be evaluated to

(lambda (x) (* x x))

and then the evaluation continues as we have described, via the substitution rule [substitute the value of the actual parameter, 2, for the formal parameter, x, everywhere x occurs in the lambda body.]

Still to discuss in connection with functions:

- → type checking
- → what is a function value?
- → can we have an operational description of function evaluation?
- → what happens if there is an undefined variable in a lambda body?
- → how can functions be composed?
- → your questions

Mentioned in class while using drracket:

- → newline
- → display

These primitives can be useful in understanding scheme - allowing simple instrumentation of our programs (eventually) and (now) of our interactions with the language.

I will summarize my use of them momentarily. first, however, it is important to know that neither of the calls

```
(newline)
(display e)
```

(where I assume e has been given a value previously) returns a value which can be input to another function.

If one attempts

```
(+ (display 1) 2)
```

for example an error will be thrown — complaining about the fact that (display 1) returns no useable value. All it does is print 1 to the screen (that's the side-effect) (*)

That being said, here is an annotated copy of an interactim with racket

```
(define x 1)          ; entered into a fresh window

(newline)             ; see what happens when
(display x)           ; you remove the (newline)s.
(newline)

((lambda (x y) (* (+ x (* y 2)) x)) 2 4)   ; opens a local
                                           ; scope for x and y
(newline)
(display x)           ; local scope has disappeared
(newline)             ; and x is again 1

(define x 17)

(newline)             ; now x is 17 in the global
(display x)           ; scope: there is no record
(newline)             ; of it ever having been 1
```

x is 2 in the local scope

Similarly, there is no record of x ever having been 2: you cannot "reach back into a scope"

What if, at the end of this session, we were to ask for a display of the value of y? An error message, complaining about y being undefined, would be shown.

What about writing just y, instead of (display y)? Same error: the first thing the special form display does is to evaluate its argument.

Note that display takes only a single argument. R5RS does not have a primitive corresponding to printf.

(*) You are familiar with side-effects in C, though it is possible that this term was not used to describe them.

Consider what you can do with a function parameter that has been passed by reference: from inside the function, you can change the ("external") value of that parameter.

Such a change is said to be a side-effect of the function call.

(**) There is no requirement that all of the formal parameters actually occur in the lambda body. Eg

```
((lambda(x y) x) 17 18)
```

makes perfect sense, and returns 17.

Indeed, there is no requirement that ANY of the formal parameters occur in the function body! Eg

((lambda (x) 3) 8) will return 3.

This is said to be a constant function.

Perhaps a better way to write such a function might be

```
(lambda () 3)
```

with calls having 0 actual parameters

```
((lambda () 3))
```

Note also that we may not write lambda forms with duplicated parameters — e.g.

```
(lambda (x x) x)
```

is not accepted by the reader.

(***) I will use standard terminology to describe parameters: in the call

```
( (lambda (x y) (+ x y)) 2 3)
```

x and y are the formal parameters and 2 and 3 are the actual parameters

(~~***~~) The number of actual parameters is set by the definition of the function, and any call with a different number of actual params than the number of formal params will be flagged as an error. This is true for C++, as well: it is a type error.