Scheme implementation

Note Title

① Currying [continues from last time – subtopic of functional forms]

Given a function of 2 or more variables, we can transform the function into a sequence of functions that have only one argument each.

This is called currying

e.g. say $f(x,y) = x^2+y$.

Define g to be a function that takes x as input, and outputs a function of one variable:

$g(x)$ = the function that accepts y as input and outputs x^2+y.

In scheme:

```
(define (f x y) (+ (* x x) y))

(define (g x)
    (lambda (y) (f x y)))
```

Then `((g x) y)` is equivalent to `(f x y)`

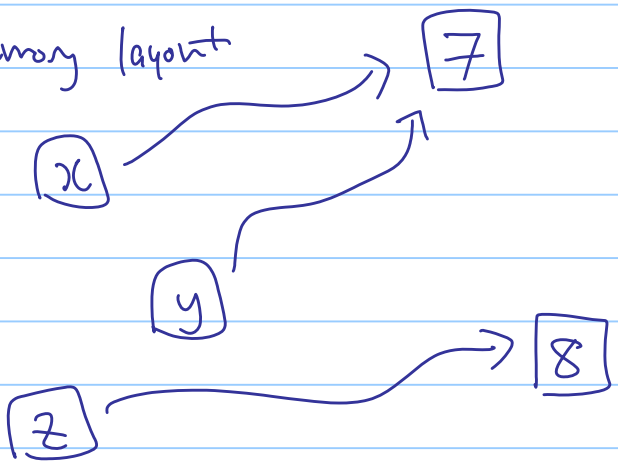
[now we start on Scheme implementation]

② Pairs, and dotted pair notation

All values in scheme are pointers to objects.

e.g.
```
(define x 7)
(define y 7)
(define z 8)
```

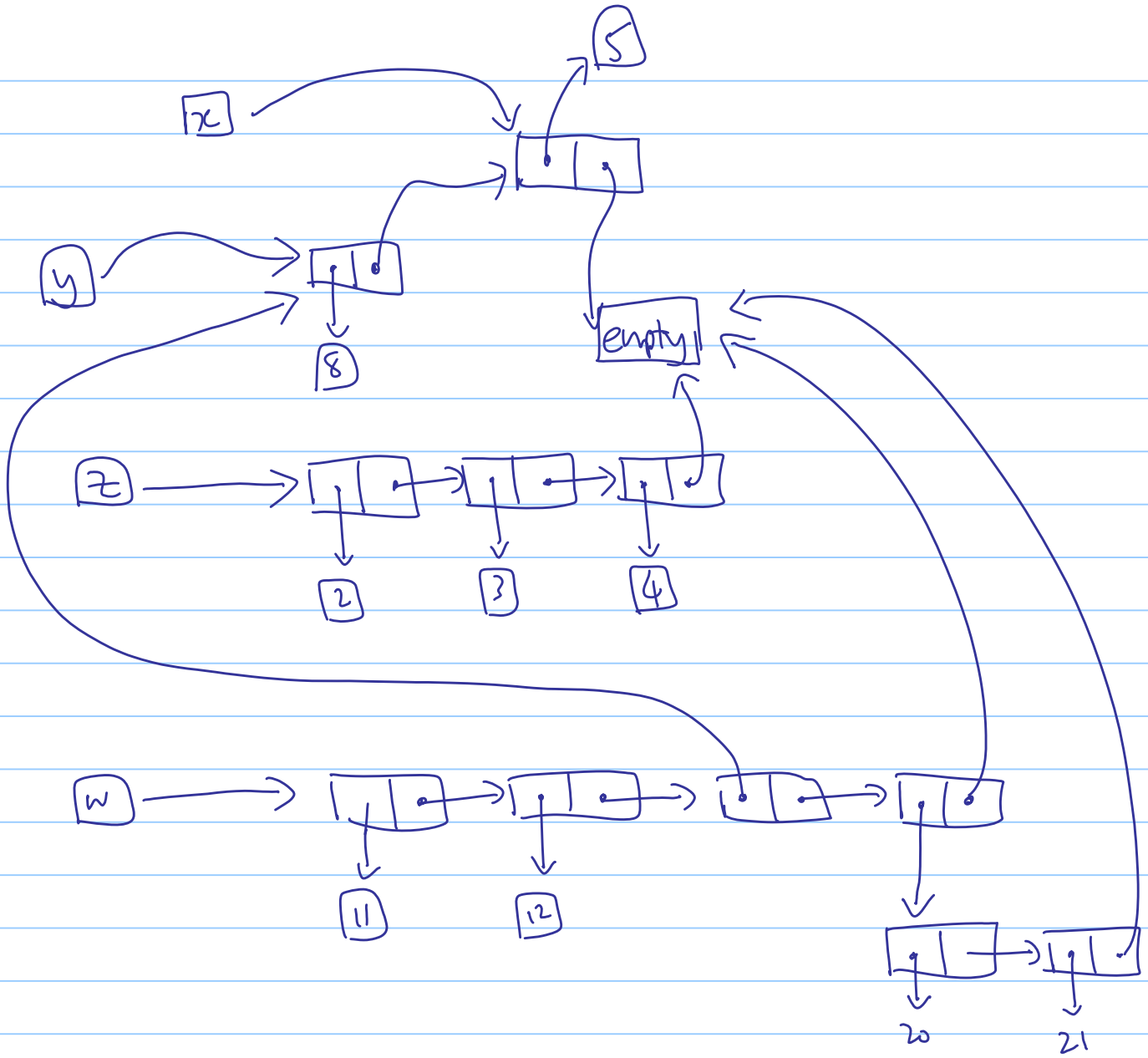
gives the memory layout



A pair is a record (think of it as a struct) containing two pointers. By definition, car returns the first pointer and cdr returns the second pointer; cons creates pairs.

e.g.
```
(define x (cons 5 empty))
(define y (cons 8 x))
(define z (list 2 3 4))
(define w (list 11 12 y (list 20 21)))
```

gives the memory layout:



Pairs are often represented in <u>dot notation</u> – the 2 elements of the pair are parenthesized and separated with a dot

e.g. x is (5 . empty)

y is (8 . (5 . empty))

z is (2 . (3 . (4 . empty)))

w is

$$(11 \cdot (12 \cdot ((8 \cdot (5 \cdot \text{empty})) \cdot (20 \cdot (21 \cdot \text{empty})))))$$

3. Is Scheme typed, or not?

All the symbols in Scheme have the type "pointer to anything".

Every object that is pointed to does have a type, which could be integer, rational, floating point, list etc..

So, Scheme does have types, and they are checked at run time, not compile time.

(This approach is known as dynamic typing.)

4. How is memory allocation handled?

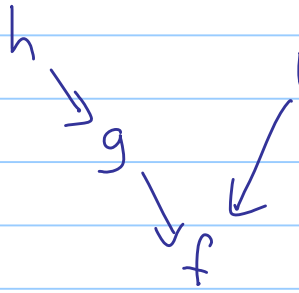
everything is automatically allocated on the heap, and automatically garbage-collected when there are no more references.

5. Strict vs lazy evaluation

Scheme (and most other languages) use strict evaluation: function arguments are evaluated before the function itself is evaluated.

e.g. `(f 3 (g (h 2) 5) 6 (p 3))`

order of evaluation is



p could be before or after h, g, but must be before f.

Lazy evaluation works the opposite way

```
e.g. (define (g x) (... some long and complex calculation))
     (define (h x) (+ x 5))
     (define (f x y z)
       (if (= x 9) (+ z x)
           (* y x)))
```

If lazy evaluation were used, then the expression

`(f (h 4) (g 6) (h 7))`

would be evaluated as:

1. begin evaluating f
2. inside the 'if' statement, we need (h 4), so begin evaluating h. This returns 9. Therefore we need '(+ z x)' – so we need (h 7). Evaluate that, returning 12. So we can return 9+12 = 21 as the value of f.

Note that the expensive function g was never evaluated, because it wasn't needed! Thus lazy evaluation can improve efficiency (but it also imposes extra costs.)

Summary of above example:

order of function evaluation

	strict	lazy
(f (h 4) (g 6) (h 7))	h, g, h, f	f, h, h
(f (h 1) (h 3) (g 2))	h, h, g, f	f, h, h, h
(f (h 1) (g 2) (h 3))	h, g, h, f	f, h, g, h

arbitrary order