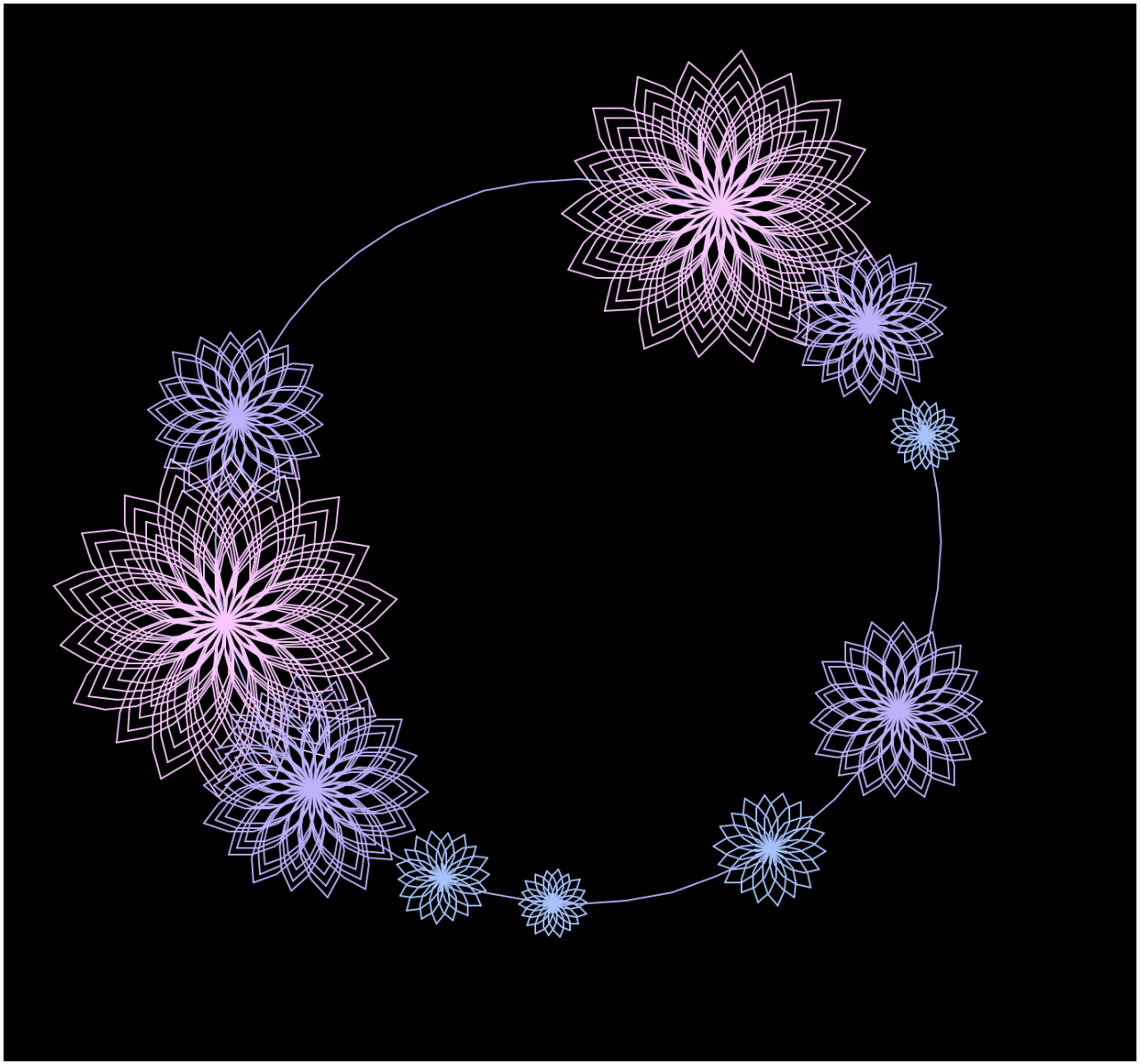
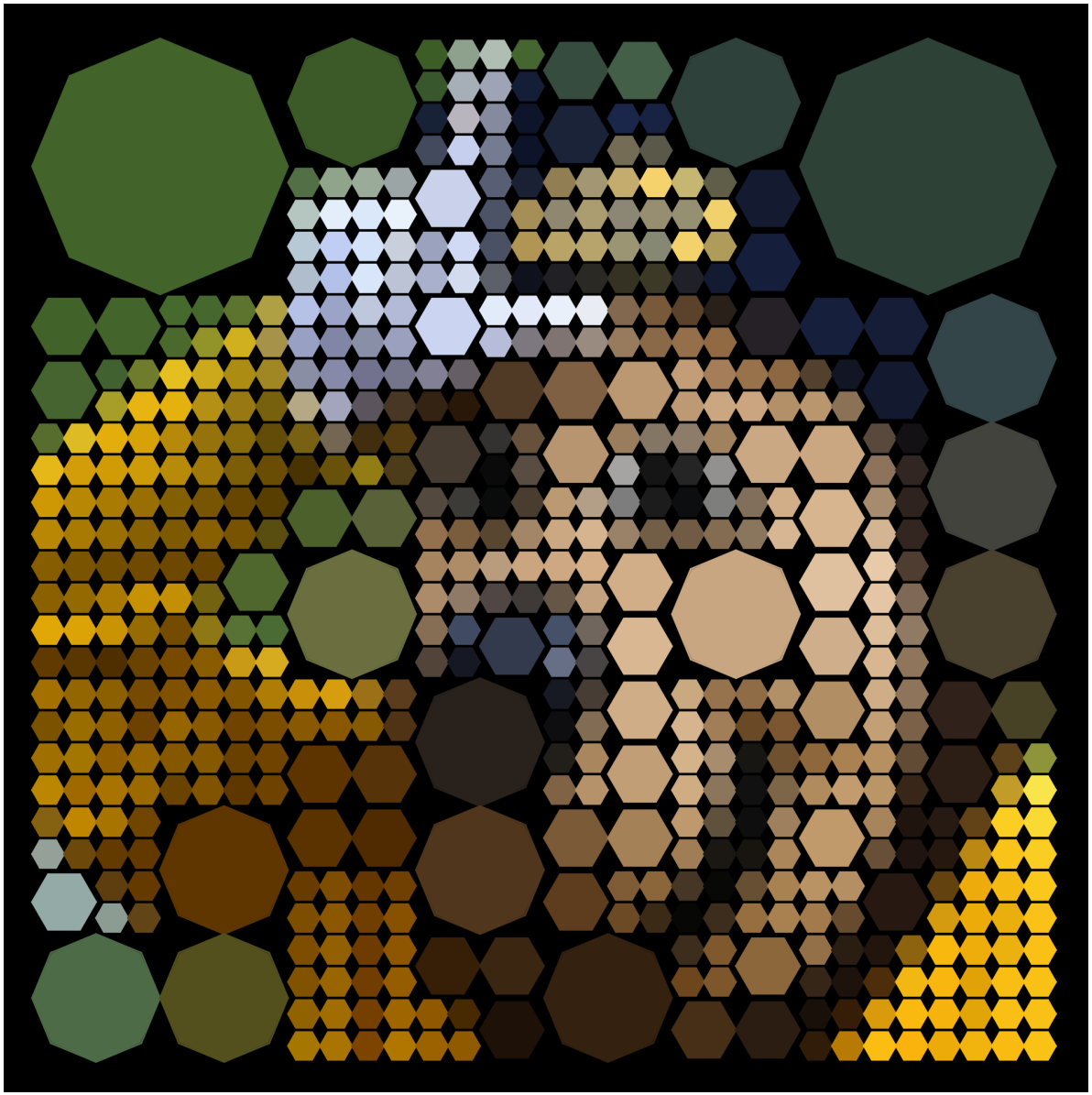
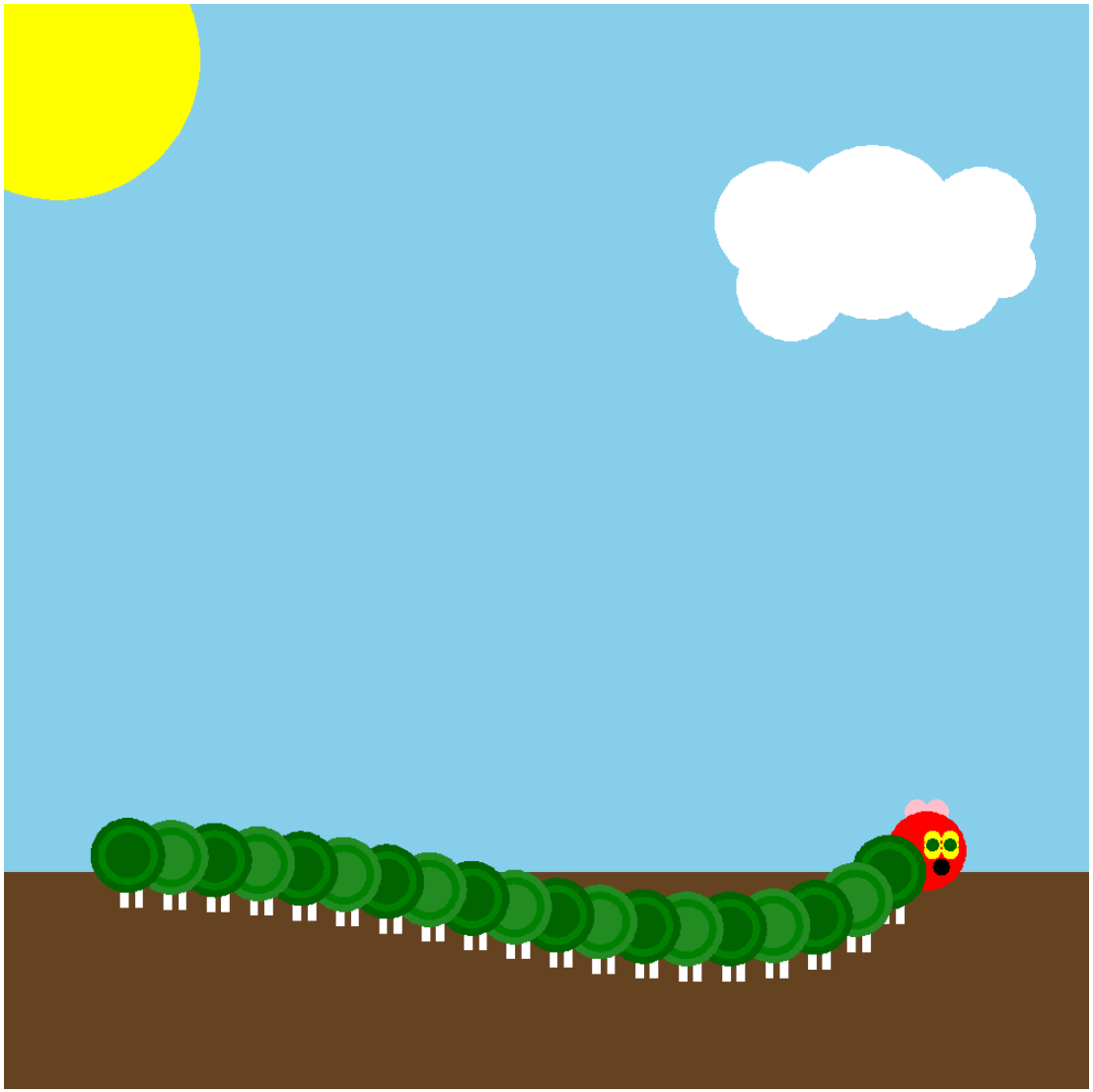
Macros

Announcements







Quasiquotation

Quasiquotation

```
There are two ways to quote an expression

  Quote:      '(a b)   =>   (a b)

  Quasiquote: `(a b)   =>   (a b)

Parts of a quasiquoted expression can be unquoted with , to evaluate sub-expressions

              (define b 4)

  Quasiquote: `(a ,(+ b 1))  =>   (a 5)

Quasiquotation is particularly convenient for generating Scheme expressions:

              (define (make-add-lambda n) `(lambda (d) (+ d ,n)))

              (make-add-lambda 2)  => (lambda (d) (+ d 2))
```

Discussion Question: Fact-Exp

Use quasiquotation to define **fact-expr,** a procedure that takes an integer n and returns a nested multiplication **expr**ession that evaluates to n **fact**orial.

```
scm> (fact-expr 5)
(* 5 (* 4 (* 3 (* 2 1))))
```

```
(define (fact-expr n)
  (if (<= n 1) 1 `(  *    ,n    ,(fact-expr (- n 1)) )))
```

`*` or `'*` or `,*`

n or **'n** or **,n**

pollev.com/cs61a

Macros

Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in a language like Lisp

61A's Scheme has a **define-macro** special form that defines a source code transformation

```
(define-macro (twice expr)
  (list 'begin expr expr))
```

```
> (twice (print 2))
2
2
```

▶ `(begin (print 2) (print 2))`

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions *without evaluating them first*
- Evaluate the expression returned from the macro procedure

(Demo)

Macros vs Procedures

A macro is an operation performed on the source code of a program before evaluation

```
(define       (second-proc expr)  (car (cdr expr)))
(define-macro (second-macro expr) (car (cdr expr)))
```

```
scm> (second-proc (list 5 7))
7
scm> (second-proc (list (+ 2 3) (+ 3 4)))
7
scm> (second-proc (list 5 7 (print 1)))
1
7
scm> (second-proc (+ 5 7))
Error: argument 0 of cdr has wrong type (int)
```

```
scm> (second-macro (list 5 7))
5
scm> (second-macro (list (+ 2 3) (+ 3 4)))
5
scm> (second-macro (list 5 7 (print 1)))
5

scm> (second-macro (+ 5 7))
5
```

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- **Call the macro** procedure on the operand expressions *without evaluating them first*
- **Evaluate the expression returned** from the macro procedure

Discussion Question: Repeat

Define **repeat**, a macro that is called on a number n and an expression expr. It evaluates expr n times, and its value is the final result.

`(repeat (+ 2 2) (print 3))` is equivalent to `(begin (print 3) (print 3) (print 3) (print 3))`

```
; Return a list containing expr n times.
; scm> (repeated-expr 4 '(print 2))
; ((print 2) (print 2) (print 2) (print 2))
(define (repeated-expr n expr)
  (if (zero? n) nil (cons expr (repeated-expr (- n 1) expr)) ))

; Evaluate expr n times and return the last value.
; scm> (repeat (+ 1 2) (print 5))  => evaluates (begin (print 5) (print 5) (print 5))
; 5
; 5
; 5
; scm> (repeat 3 (+ 2 3))  ; (+ 2 3) is evaluated 3 times, but only the last is returned
; 5
(define-macro (repeat n expr)
  ( cons  'begin (repeated-expr (eval n)  expr  )))
```

pollev.com/cs61a

Discussion Question: Repeat Repeat

Define repeat, a macro that is called on a number n and an expression expr. It evaluates expr n times, and its value is the final result.

```
(repeat (+ 2 2) (print 3)) is equivalent to:
(begin
  (define (repeater k)
    (if (= k 1) (print 3) (begin (print 3) (repeater (- k 1)))))
  (repeater 4))
```

```
; Return an expression that will repeatedly evaluate expr n times using recursion.
; scm> (repeated-expr 4 '(print 2))
; ((define (repeater k) (if (= k 1) (print 2) (begin (print 2) (repeater (- k 1))))) (repeater 4))
(define (repeated-expr n expr)
  `(define (repeater k)
     (if (= k 1) ,expr (begin ,expr (repeater (- k 1)))))
   (repeater ,n) ))
```

pollev.com/cs61a

```
; Evaluate expr n times and return the last value.

(define-macro (repeat n expr)
  (cons 'begin (repeated-expr (eval n) expr )))
```

For Macro

For Macro

Define a `for` macro that evaluates an expression for each value in a sequence

```
scm> (for x '(2 3 4 5) (* x x))
(4 9 16 25)

scm> (map (lambda (x) (* x x)) '(2 3 4 5))
(4 9 16 25)

(define-macro (for sym vals expr)
  (list 'map (list 'lambda (list sym) expr) vals))
```

Rewrite it using quasiquotation

```
(define-macro (for sym vals expr)
  `( map ( lambda ( ,sym ) ,expr ) ,vals ))
```

Why not define it so that the values don't need to be quoted?

```
scm> (for x (2 3 4 5) (* x x))
(4 9 16 25)
```