#lang racket

; Cayley Map Face Generator

; Maddie Spies

; November 2019

;DESCRIPTION

;When given a group and rotation, this program returns the group, the rotation, the faces that are formed from the Cayley Map of the specified rotation,

;the size and number of each face, the total number of faces, the Cayley Map's Euler characteristic and genus.

;INSTRUCTIONS

;If not already represented in leters, convert your group from numbers to letters. (For example, (0 1) is (b) and (2 3) is (a a b b b))

;Input the group under the GROUP DEFINITION section and define it as G. The first part of G should be each element (a,b, etc.) in the group and its inverse.

;The second part of G defines the identity elements (how many a's you need to get null, how many b's you need to get null, etc.) as well as any other

;information that could be useful when simpifying the elements (for example, what (b a) is equivalent to).

;The third part of G is how many vertices are in the complete bipartite graph that the group is describing.

;The fourth part of G is the name of the group and the identity elements (in number form, if applicable).

;Template: (define G '( (elements and their inverses) (identities and other equivalencies) # of vertices ("group name"))

;Input the rotation under the ROTATION DEFINITION section and define it as r. Write each element in the rotation in ().

;Template:(define r '((1st element of rotation)(2nd element)(3rd element)(etc.)))

;GROUP DEFINITION

;A group is a list of two items, the first is a list of generators and their inverses, the second is a list of relations,

;the third is the number of vertices, and the fourth is the name of the group.

;(define G '(( (a (a a a a a)) (b (b)) ) (((a a a a a a) ()) ((b b) ()) ((b a) ( a a a a a b))) 12 ("D6 (6 0) (0 2)"))) ;D6

;(define G '(( (r (r r)) (f (f)) ) (((r r r) ()) ((f f) ()) ((f r) (r r f))) 6 ("D3 (3 0) (0 2)"))) ;D3

;(define G '(( (r (r r r r)) (f (f)) ) (((r r r r r) ()) ((f f) ()) ((f r) (r r r r f))) 10 ("D5 (5 0) (0 2)"))) ;D5

(define G '(( (r (r r r r r r)) (f (f)) ) (((r r r r r r r) ()) ((f f) ()) ((f r) (r r r r r r f))) 14 ("D5 (7 0) (0 2)"))) ;D7

;(define G '(( (a (a)) (b (b)) (c (c)) ) (((a a) ()) ((b b) ()) ((c c) ()) ((b a) (a b)) ((c a) (a c)) ((c b)(b c))) 8 ("Z2 x Z2 x Z2"))) ;Z2 x Z2 x Z2;

;(define G '(( (a (a a a a a a a)) ) (((a a a a a a a a) ())) 8 ("Z8 (8)"))) ;Z8

;(define G '(( (a (a)) (b (b b b b b)) ) (((a a) ()) ((b b b b b b) ()) ((b a) (a b))) 12 ("Z3 x Z6 (3 4)"))) ;Z2 x Z6

;(define G '(( (a (a a)) (b (b b b)) ) (((a a a) ()) ((b b b b) ()) ((b a) (a b))) 12 ("Z3 X Z4 (3 0) (0 4)"))) ;Z3 x Z4

;(define G '(( (a (a a a a a)) ) (((a a a a a a) ())) 6 ("Z6 (6)"))) ;Z6

;(define G '(( (a (a a a a a a a a a)) ) (((a a a a a a a a a a) ())) 10 ("Z10 (5)"))) ;Z10

;(define G '(( (a (a a a a a a a a a a a)) ) (((a a a a a a a a a a a a) ())) 12 ("Z12 (12)"))) ;Z12

;(define G '(( (a (a a a a a a a a a a a a a)) ) (((a a a a a a a a a a a a a a) ())) 14 ("Z14 (14)"))) ;Z14

;(define G '(( (a (a a a a a a a a a a a a a a a)) ) (((a a a a a a a a a a a a a a a a) ())) 16 ("Z16 (16)"))) ;Z16;

;(define G '(( (a (a a a a a a a a a a a a a a a a a)) ) (((a a a a a a a a a a a a a a a a a a) ())) 18 ("Z18 (18)"))) ;Z18

;(define G '(( (a (a a a a a a a a a a a a a a a a a a a a a)) ) (((a a a a a a a a a a a a a a a a a a a a a a) ())) 22 ("Z22 (22)"))) ;Z22

;(define G '(( (a (a a a a a a a a a a a a a a a a a a a a a a a a a)) ) (((a a a a a a a a a a a a a a a a a a a a a a a a a a) ())) 26 ("Z26 (26)"))) ;Z22

;ROTATION DEFINITION

;(define r '((a) (a a a b) (a a a a a b) (a a a a a)(a b) (a a a))) ;D6

;(define r '( (r f)(f) (r r r f) (r r f) (r r r r f) )) ;D5

;(define r '( (f) (r f) (r r f) )) ;D3

(define r '( (r f) (r r r f) (r r r r f) (r r r r r f) (r r f) (r r r r r r f) (f))) ;D7

;(define r '( (c) (a) (a b) (b c))) ;Z2 x Z2 x Z2

;(define r '( (a a a a a a a) (a) (a a a a a) (a a a))) ;Z8

;(define r '( (a a a) (a a a a a a a) (a a a a a) (a))) ;Z8

;(define r '((a a b b) (a b b) (a a b) (a b b b) (b) (b b b))) ;Z2 x Z6

;(define r '( (a b) (a a b) (a b b b) (b) (b b b) (a a b b b) )) ;Z3 x Z4

;(define r '( (a a a a a) (a a a) (a))) ;Z6

;(define r '( (a) (a a a a a a a a a) (a a a a a) (a a a) (a a a a a a a ) )) ;Z10

;(define r '( (a) (a a a) (a a a a a) (a a a a a a a a a a a) (a a a a a a a a a)(a a a a a a a )));Z12

;(define r '( (a a a a a) (a) (a a a) (a a a a a a a) (a a a a a a a a a) (a a a a a a a a a a a a a) (a a a a a a a a a a a))) ;Z14;

;(define r '( (a) (a a a a a a a a a a a a a a a) (a a a a a a a a a a a a a) (a a a a a a a a a a a) (a a a a a a a a a) (a a a a a a a ) (a a a a a) (a a a)) );Z16

;(define r '((a a a a a a a a a a a a a) (a a a a a) (a) (a a a) (a a a a a a a a a) (a a a a a a a) (a a a a a a a a a a a a a a a a a) (a a a a a a a a a a a a a a a a a a a) (a a a a a a a a a a a a a a a a a a a a a) (a a a a a a a a a a a) (a a a a a a a a a a a a a a a) )) ;Z22

;(define r '((a a a a a a a) (a a a a a) (a a a a a a a a a a a a a a a a a) (a a a a a a a a a a a a a a a a a a a a a a a) (a a a a a a a a a a a) (a a a a a a a a a a a a a a a a a a a a a a a a a) (a a a a a a a a a a a a a a a a a a a) (a a a a a a a a a a a a a)(a a a a a a a a a) (a a a a a a a a a a a a a a a a a a a a a) (a) (a a a a a a a a a a a a a a a) (a a a) )) ;Z26

; Both word and pattern are lists.

; If word begins with pattern, return #t. If not, return #f.

(define (beginsWith? word pattern)

(cond

((null? pattern) #t)

((null? word) #f)

(else (and (equal? (first word) (first pattern)) (beginsWith? (rest word) (rest pattern))))))

;(beginsWith? word (first (third (second G)))

;Reverse is list of pairs and we are comparing reverse of first item of each pair to beginning of word

(define (beginsWithAny? word relationList)

(cond

((null? relationList) #f)

((beginsWith? word (reverse (first (first relationList)))) (first relationList))

(else (beginsWithAny? word (rest relationList)))))

;(beginsWithAny? word (second G))

;Returns #t if word contains element in second G

(define (match? temp word relationList)

(cond

((beginsWithAny? temp relationList) (list temp word (beginsWithAny? temp relationList)))

((null? word) #f )

(else (match? (cons (first word) temp) (rest word) relationList))))

;(match? '() word (second G))

;Removes n elements from word and returns word without those n elements

(define (removeFromFront word n)

(cond

((= n 0) word)

((null? word) "Oops, you tried to remove too many things from a list in removeFromFront")

(else (removeFromFront (rest word) (sub1 n)))))

;(removeFromFront '(a a a a a a b a b a) 22)

;Removes n elements from front of word and puts new elements on the front of the word

(define (replace word n new)

(append new (removeFromFront word n)))

;(replace '(a a a a a a b a b a) 6 '(b a b b))

;Uses second G relations to simplify word and returns simplified word

(define (simplify word relationList)

(cond

((null? relationList) word)

(else

(let ([match (match? '() word relationList)])

(cond

((not match) word)

(else

(let ([temp (first match)] [word (second match)] [n (length (first (third match)))] [newExp (reverse (second (third match)))])

(simplify (append (reverse (replace temp n newExp)) word) relationList))))))))

;(simplify '(a a a a a a b b b b b a b a b a b) (second G))

;Returns simplified element

(define (simplifyElement word)

(simplify word (second G)))

;(simplifyElement '(a a a a a a b b b b b a b a b a b))

(define (relInverse generator inverseList)

(cond

((null? inverseList) "That was not a generator")

((equal? generator (first (first inverseList))) (second (first inverseList)))

(else (relInverse generator (rest inverseList)))))

;(relInverse 'b inverseList)

;Returns inverse of element

(define (genInverse generator)

(relInverse generator (first G)))

;(genInverse 'a)

;Returns simplified inverse of element. tempInverse should be null list

(define (inverse element tempInverse)

(cond

((null? element) (simplifyElement tempInverse))

(else (inverse (rest element) (append (genInverse (first element)) tempInverse)))))

;(inverse '(b a) '())

;returns next element in rotation; returns "element not in rotation" if you input an element not in rotation

(define (nextElement element rotation)

(cond

((null? rotation) "error- null rotation")

((equal? element (last rotation)) (first rotation))

((member element r) (second (member element r)))

(else "element not in rotation")))

;(nextElement '(a b) r)

;Returns next edge; returns "element not in rotation" if you input an element not in rotation

(define (getNextEdge element rotation)

(cond

((equal? (nextElement (inverse element '()) rotation) '"element not in rotation") #f)

(else (nextElement (inverse element '()) rotation))))

;(getNextEdge '(a a a a a b) r)

;Returns 1 face

(define (getFace face rotation)

(cond

((equal? (first face) (getNextEdge (last face) rotation)) face)

(else (getFace (append face (list (getNextEdge (last face) rotation))) rotation))))

;(getFace '((b)) r)

;Returns #f if element is not in faceList, otherwise it returns the element and the other elements that are in the face that come after the element

(define (inFaceList? element faceList)

(cond

((null? faceList) #f)

(else (or (member element (first faceList)) (inFaceList? element (rest faceList))))))

;(inFaceList? '(b a) '(((b)(b a) (a a b)) ((a a a b) (b b b))))

;Returns element in rotation thats not in faceList

(define (nextNewElement faceList rotation)

(cond

((null? rotation) #f)

((inFaceList? (first rotation) faceList) (nextNewElement faceList (rest rotation)))

(else (first rotation))))

;(nextNewElement '() r)

;Returns list of all the faces

(define (getAllFaces faceList rotation)

(cond

((not (nextNewElement faceList rotation)) faceList)

(else (getAllFaces (cons (getFace (list (nextNewElement faceList rotation)) rotation) faceList) rotation))))

;(getAllFaces '() r)

;Returns number of generating letters in element. temp=0

(define (count element generator temp)

(cond

((null? element) temp)

((equal? (first element) generator) (count (rest element) generator (+ 1 temp)))

(else (count (rest element) generator temp))))

;(count '(a a a b b) 'a 0)

;Returns a b element as a pair of numbers

(define (convert element generatorList)

(cond

((null? generatorList) '())

(else (cons (count element (first (first generatorList)) 0) (convert element (rest generatorList))))))

;(convert '(a a b a b b) (first G))

;Returns face of a b elements as a list of pairs of numbers

(define (convertFace face generatorList)

(cond

((null? face) '())

(else (cons (convert (first face) generatorList) (convertFace (rest face) generatorList)))))

;(convertFace '((a b) (b b b) (a b b)) (first G))

;Returns list of letters as numbers

(define (convertList list generatorList)

(cond

((null? list) '())

(else (cons (convertFace (first list) generatorList) (convertList (rest list) generatorList)))))

;(convertList (getAllFaces '() r) (first G))

;Returns the order of an element

(define (order element currentProduct currentOrder)

(cond

((null? (simplify currentProduct (second G))) currentOrder)

(else (order element (simplify (append element currentProduct) (second G)) (+ 1 currentOrder)))))

;(order '(a) '(a) 1)

;Returns face as single list

(define (makeFaceList face tempFace)

(cond

((null? face) tempFace)

(else (makeFaceList (rest face) (append (first face) tempFace)))))

;(makeFaceList '((a a) (b b) (c c)) '())

;Returns the order of a face

(define (orderFace face)

(order (makeFaceList face '()) (makeFaceList face '()) 1))

;(orderFace '((a b b) (a) (b)))

;Returns how many sides face has

(define (sizeOfFace face)

(\* (length face) (orderFace face)))

;(sizeOfFace '((a b b)(a) (b)))

;Returns the number of a specified face

(define (numberOfAFace face)

(/ (third G) (orderFace face)))

;(numberOfAFace '((a b b) (a) (b)))

;Returns table header

(define (tableHeader )

(begin

(display "Face Size" )(display " | ") (display "Number of Faces")

(display "\n")))

;(tableHeader)

;Returns body of table (with faceSize and numberOfAFace)

(define (table faceList)

(cond

((null? faceList)(display "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"))

((> (sizeOfFace (first faceList)) 9) (display (sizeOfFace (first faceList)))(display " | ") (display (numberOfAFace (first faceList)))(display "\n")

(table (rest faceList)))

(else (display (sizeOfFace (first faceList)))(display " | ") (display (numberOfAFace (first faceList)))(display "\n")

(table (rest faceList)))))

;(table (getAllFaces '() r))

;Returns total number of faces

(define (faceTotal faceList temp)

(cond

((null? faceList) temp)

(else (faceTotal (rest faceList) (+ (numberOfAFace (first faceList)) temp) ))))

;(faceTotal (getAllFaces '() r) 0)

;returns euler char. (v-e+f)

(define (eulerChar vertices edges faces)

(+ (- vertices edges) faces))

;(eulerChar (third G) (\* (/ (third G) 2) (/ (third G) 2)) (faceTotal (getAllFaces '() r) 0))

;returns genus

(define (genus)

(/ (- 2 (eulerChar (third G) (\* (/ (third G) 2) (/ (third G) 2)) (faceTotal (getAllFaces '() r) 0))) 2))

;(genus)

;Displays a list of things

(define (displayAll aList)

(if (null? aList) (display "")

(begin

(display (first aList))

(display " ")

(displayAll (rest aList)))))

;(displayAll '("aa" "bb" 12))p

;Returns the table header, group, rotation, faces, table with face sizes and numbers, toatl numbe rof faces, euler characteristic, and genus

(define (makeTable)

(begin (display "Group: ")(display (first (fourth G)) )

; (display " ")(display (convertList (second G) (first G)))

(display "\n"))

(begin (display "Rotation: ")(display (convertList (list r) (first G))))

(begin (display "\n") (display "Faces: ")(display (convertList (getAllFaces '() r) (first G)))

(display "\n") (display "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_") (display "\n")

(tableHeader)(table (getAllFaces '() r))(display "\n")

(display (faceTotal (getAllFaces '() r) 0)) (display " faces")

(display "\n")(display "Euler Char.:X=")

(display (eulerChar (third G) (\* (/ (third G) 2) (/ (third G) 2)) (faceTotal (getAllFaces '() r) 0)))

(display "\n")(display "Genus: g=")(display (genus))))

(makeTable)