1 Laying Bricks:Getting the Computer to do Repetitive Tasks

1.1 Working with expression sequences and lists

The process of analyzing or solving a problem can involve a lot of repetition. Anyone who has graphed many different functions knows the tedious repetition of that process. Maple's built-in "plot" commands allow the computer do the mindless repetition, freeing time for people to look at the relationships implied by the graph - something the machine can't do. Graphing, however is only one example of this of this capability which is absolutely critical to effective use of the computer for solving general problems. Lets look at another - building lists.

We have already made and used a lot of Maple lists. A more primitive notion than that of a list is that of an expression sequence

1 *Definition: An expression sequence is a comma-separated expression of the form "a,b,c,...,z" where the elements "a", "b" , etc. can be any symbols recognizable by Maple.*

The order of elements in an expression sequence is important: "a,b" is a different expression sequence from "b,a"

For our immediate purposes the most important thing about expression sequences is how easy it is to extend them. If joe is an expression sequence and we want to add "Hortense" to the end of joe we can simply enter joe:=joe, Hortense; . If we wanted to place Hortense at the beginning we would enter joe:= Hortense,joe; This idea also allows us to "build" sequences.

If we initially have a sequence named "joe" and need, in the course of a calculation to extend "joe" then this can be done through steps like

```
joe:=I;
```

$$joe := I$$

```
joe:=joe,like;
```

$$joe := I,\ like$$

```
joe:=joe,math;
```

Sometimes it is useful as a starting point to have an empty expression. That is something which is an expression but has no items in it. This is evidently analogous to the empty set in set theory. This idea is so useful that Maple has a special name for the emptyexpression sequence. It is called NULL....

$$joe := I,\ like,\ math$$

```
joe:=NULL;
```

$$joe :=$$

```
joe:=joe,I;
```

$$joe := I$$

```
joe:=joe,like;
```

$$joe := I,\ like$$

```
joe:=joe,math;
```

$$joe := I,\ like,\ math$$

With the notion of expression sequence we can observe that a list is simply an expression sequence delimited by brackets "[ ]". Most important, to convert an expression sequence, joe to a list sam we simply enclose it in brackets

```
sam:=[joe];
```

$$sam := [I,\ like,\ math]$$

Since a list is simply an expression sequence enclosed in brackets there is a simple way to extract the undellying expression sequence from a list with a command called **op.** We will encounter op quite often. This is but one of its many uses. (op **stands for operands)**

1.2 How to Build a list by hand

Maple has a word **whattype** which we can use to ask what type it has assigned to various objects (there are many kinds of types). Now let's illustrate.

First let's make a simple sequence. We will start with the NULL sequence to indicate how it is used as a starting point for building sequences

```
joe:=NULL;
```

$$joe :=$$

```
harry:=joe,sally,1;
```

$$harry := sally,\ 1$$

```
whattype(joe);
```

$$exprseq$$

```
whattype(harry);
```

$$exprseq$$

```
joe:= harry,dog, Pi;
```

$$joe := sally,\ 1,\ dog,\ \pi$$

```
whattype(joe);
```

$$exprseq$$

```
sam:=[joe];
```

$$sam := [sally,\ 1,\ dog,\ \pi]$$

```
whattype(sam);
```

$$list$$

```
tim:=op(sam);
```

$$tim := sally,\ 1,\ dog,\ \pi$$

```
whattype(tim);
```

$$exprseq$$

Now suppose we want to make the list stuff:= [sally,1,dog, π, Lance]; . We observe that we should be able to do this simply by adding "Lance" to the end of the list "sam". We would like to do something like stuff:=sam,Lance; **or stuff:=[sam,Lance]; but neither of these will work. This is not to say that Maple won't accept the commands**

```
stuff:=sam,Lance;
```

$$stuff := [sally,\ 1,\ dog,\ \pi],\ Lance$$

```
whattype(stuff);
```

$$exprseq$$

```
stuff:=[sam,Lance];
```

$$stuff := [[sally,\ 1,\ dog,\ \pi],\ Lance]$$

Neither of these works. The first produces an expression sequence, not a list. Moreover it has only two elements, the first being the list "sam" and and the second the expression "Lance". The second does produce a list but it is a list with two elements: the first element of this is the list " [sally,1,dog, π]" and the second is "Lance".

In order to produce the list we want we have to extract the expression sequence from "sam", append "Lance " to it and then convert that sequence to a list.

```
tmp:=op(sam);
```

$$tmp := sally,\ 1,\ dog,\ \pi$$

```
tmp:=tmp,Lance;
```

$$tmp := sally,\ 1,\ dog,\ \pi,\ Lance$$

```
stuff:=[tmp];
```

$$stuff := [sally,\ 1,\ dog,\ \pi,\ Lance]$$

```
whattype(stuff);
```

$$list$$

We can be more economical by combining steps by composing commands:

```
stuff:=[op(sam),Lance];
```

$$stuff := [sally,\ 1,\ dog,\ \pi,\ Lance]$$

Incidentally, to go with "op" there is a very useful commnad **nops (number of operands)** which we can use to tell us how many items there are in a list

```
junk:=[a,b,cow];
```

$$junk := [a, b, cow]$$

```
nops(junk);
```

$$3$$

Note that nops doesn't work on an expression sequence.

```
nops(NULL);
```

```
Error, wrong number (or type) of parameters in function nops
```

```
sally:=1,2,3;
```

$$sally := 1, 2, 3$$

```
nops(sally);
```

```
Error, wrong number (or type) of parameters in function nops
```

Thus in order to count the number of items in an expression sequence we must ask nops to act on a **list** of which the expression sequence is the operand.

```
sally:=1,2,3;
```

$$sally := 1, 2, 3$$

```
nops([sally]);
```

$$3$$

```
whattype(sally);
```

$$exprseq$$

It is important to note that writing "[sally]" above does not change "sally" into a list. The **assignment** command sally:=[sally]; would do that but above we have not assigned a new value to "sally". All we have done is ask nops to act on an unnamed list constructed from "sally". This leads to a comment we must make regarding what happens when we make an expression sequence some of whose elements are objects that have assigned values. For instance suppose we have executed bird:=turkey; and subsequently make an expressions sequence animals:=cat,dog,bird; The sequence animals will then be cat,dog,turkey. That is, the value that has been assigned to any of the terms in the sequence we write down will be substituted immediately. If we then redefine "bird", say bird:=pigeon; the sequence animals will still be cat,dog,turkey unless we do something which re-defines animals (e.g. we might re-execute the command animals:=cat,dog,bird; at which time animals will become cat,dog,pigeon **.**

```
bird:=turkey;
```

$$bird := turkey$$

```
animals:=cat,dog,bird;
```

$$animals := cat,\ dog,\ turkey$$

```
print([animals]);
```

$$[cat,\ dog,\ turkey]$$

```
bird:=pigeon;
```

$$bird := pigeon$$

```
print([animals]);
```

$$[cat,\ dog,\ turkey]$$

```
animals:=cat,dog,bird;
```

$$animals := cat,\ dog,\ pigeon$$

```
print([animals]);
```

$$[cat,\ dog,\ pigeon]$$

This property of sequences is extremely important for it allows us to build a sequence in steps re-using a temporary name for the item we are adding.

```
dogs:=NULL;
```

$$dogs :=$$

```
tmpdog:=Beagle;
```

$$tmpdog := Beagle$$

```
dogs:=dogs,tmpdog;
```

$$dogs := Beagle$$

```
tmpdog:=Airdale;
```

$$tmpdog := Airdale$$

```
dogs:=dogs,tmpdog;
```

$$dogs := Beagle,\ Airdale$$

1.3 Automatically Generating Sequences and Lists

Now suppose we wanted to make a list of the first 25 positive intgers. One way to do this is simply to enter it:

```
numlist:=[1,2,3,4,5,6,7,8,9,10,11,
12,13,14,15,17,17,18,19,20,21,22,23,24,24,25];
```

$$numlist := [1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14,\ 15,\ 17,\ 17,\ 18,\ 19,\ 20,\ 21,\ 22,\ 23,\ 24,\ 24,\ 25]$$

This is tedious and fraught with opportunity to make mistakes. Indeed the careful reader will have noted that this isn't the correct list, "17" was entered twice. Even that careful soul might not take the time to check whether this sequence which clams to list the numbers from 672345 to 677345 is actually correct.

1.3.1 Boring List of Numbers from 672345 To 677345

boringlist := [672345, 672346, 672347, 672348, 672349, 672350, 672351, 672352, 672353, 672354,
672355, 672356, 672357, 672358, 672359, 672360, 672361, 672362, 672363, 672364, 672365, 672366,
672367, 672368, 672369, 672370, 672371, 672372, 672373, 672374, 672375, 672376, 672377, 672378,
672379, 672380, 672381, 672382, 672383, 672384, 672385, 672386, 672387, 672388, 672389, 672390,
672391, 672392, 672393, 672394, 672395, 672396, 672397, 672398, 672399, 672400, 672401, 672402,
672403, 672404, 672405, 672406, 672407, 672408, 672409, 672410, 672411, 672412, 672413, 672414,
672415, 672416, 672417, 672418, 672419, 672420, 672421, 672422, 672423, 672424, 672425, 672426,
672427, 672428, 672429, 672430, 672431, 672432, 672433, 672434, 672435, 672436, 672437, 672438,
672439, 672440, 672441, 672442, 672443, 672444, 672445, 672446, 672447, 672448, 672449, 672450,
672451, 672452, 672453, 672454, 672455, 672456, 672457, 672458, 672459, 672460, 672461, 672462,
672463, 672464, 672465, 672466, 672467, 672468, 672469, 672470, 672471, 672472, 672473, 672474,
672475, 672476, 672477, 672478, 672479, 672480, 672481, 672482, 672483, 672484, 672485, 672486,
672487, 672488, 672489, 672490, 672491, 672492, 672493, 672494, 672495, 672496, 672497, 672498,
672499, 672500, 672501, 672502, 672503, 672504, 672505, 672506, 672507, 672508, 672509, 672510,
672511, 672512, 672513, 672514, 672515, 672516, 672517, 672518, 672519, 672520, 672521, 672522,
672523, 672524, 672525, 672526, 672527, 672528, 672529, 672530, 672531, 672532, 672533, 672534,
672535, 672536, 672537, 672538, 672539, 672540, 672541, 672542, 672543, 672544, 672545, 672546,
672547, 672548, 672549, 672550, 672551, 672552, 672553, 672554, 672555, 672556, 672557, 672558,
672559, 672560, 672561, 672562, 672563, 672564, 672565, 672566, 672567, 672568, 672569, 672570,
672571, 672572, 672573, 672574, 672575, 672576, 672577, 672578, 672579, 672580, 672581, 672582,
672583, 672584, 672585, 672586, 672587, 672588, 672589, 672590, 672591, 672592, 672593, 672594,
672595, 672596, 672597, 672598, 672599, 672600, 672601, 672602, 672603, 672604, 672605, 672606,
672607, 672608, 672609, 672610, 672611, 672612, 672613, 672614, 672615, 672616, 672617, 672618,
672619, 672620, 672621, 672622, 672623, 672624, 672625, 672626, 672627, 672628, 672629, 672630,
672631, 672632, 672633, 672634, 672635, 672636, 672637, 672638, 672639, 672640, 672641, 672642,
672643, 672644, 672645, 672646, 672647, 672648, 672649, 672650, 672651, 672652, 672653, 672654,
672655, 672656, 672657, 672658, 672659, 672660, 672661, 672662, 672663, 672664, 672665, 672666,
672667, 672668, 672669, 672670, 672671, 672672, 672673, 672674, 672675, 672676, 672677, 672678,
672679, 672680, 672681, 672682, 672683, 672684, 672685, 672686, 672687, 672688, 672689, 672690,
672691, 672692, 672693, 672694, 672695, 672696, 672697, 672698, 672699, 672700, 672701, 672702,
672703, 672704, 672705, 672706, 672707, 672708, 672709, 672710, 672711, 672712, 672713, 672714]

"boringlist" is actually correct because it was not typed in by hand but rather done by a Maple command **seq** which generates expression sequences according to a "formula" we prescribe which tells Maple how to find the "nth" element. A few examples suffice to illustrate this very useful command. First, here is the command that generated "boringlist"

```
boringlist:=[seq(i,i=672345..672714)];
```

Note that seq returns an expression sequence. The brackets in turn make that a list. An expression sequence of the cubes of the integers from -3 to 13 is simply

```
cubesequence:=seq(i^3,i=-3..13);
```

$$cubesequence := -27, -8, -1, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197$$

We can convert it into a list

```
cubelist:=[cubesequence];
```

$$cubelist := [-27, -8, -1, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197]$$

if the list was all we wanted we could have skipped naming the sequence and simply entered

```
cubelist:=[seq(i^3,i=-3..13)];
```

$$cubelist := [-27, -8, -1, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197]$$

Some further examples of lists:

Make a list if the odd integers from 1 to 99

```
oddlist:=[seq( 2*j+1,j=0..49)];
```

$$oddlist := [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99]$$

Make an expression sequence of all the natural numbers less than 137 which are 2 plus a multiple of 7.

```
twomodsevenseq:=seq(2+7*q,q=0..137/7);
```

$$twomodsevenseq := 2, 9, 16, 23, 30, 37, 44, 51, 58, 65, 72, 79, 86, 93, 100, 107, 114, 121, 128, 135$$

Make a list of lists of numbers so that the nth element in your list is the list $[n, \frac{1}{n^2}]$, starting with n=10 and ending with n=30.

```
listoflists:=[seq([n,1/n^2],n=10..30)];
```

$$listoflists := [[10, \frac{1}{100}], [11, \frac{1}{121}], [12, \frac{1}{144}], [13, \frac{1}{169}], [14, \frac{1}{196}], [15, \frac{1}{225}], [16, \frac{1}{256}], [17, \frac{1}{289}], [18, \frac{1}{324}], [19, \frac{1}{361}], [20, \frac{1}{400}], [21, \frac{1}{441}], [22, \frac{1}{484}], [23, \frac{1}{529}], [24, \frac{1}{576}], [25, \frac{1}{625}], [26, \frac{1}{676}], [27, \frac{1}{729}], [28, \frac{1}{784}], [29, \frac{1}{841}], [30, \frac{1}{900}]]$$

1.4 Exercises: Sequences and Lists

1.4.1 Exercise: Make a list of all the natural numbers which are less than 1000 and are the square of another natural number.

1.4.2 Exercise: *Make a sequence of length 100 such that the jth element of list whose first element is the integer j and who second element is the factorization of j into primes.*

1.4.3 Exercise: *Make a list of length 5 whose jth element is a plot of the function sin(j*x) on the interval [0,* 2π*]. Display all of the graphs at once with plots[display].*

1.4.4 Exercise: *Make a sequence of 100 random integrers between 0 and 10. (If necessary, use "help" to look up the topic "random").*

1.4.5 Exercise: If "n" is a natural number then n factorial, denoted "n!" is the product n(n-1)(n-2)... (3)(2)(1). For instance 3!=(3)(2)(1) = 6, 1! = 1, 4!=(4)(3)(2)(1) = 24. Maple recognizes the notation n! and will compute the factorial of a natural number. Make a list of the factorials of the numbers from 1 to 20.

1.4.6

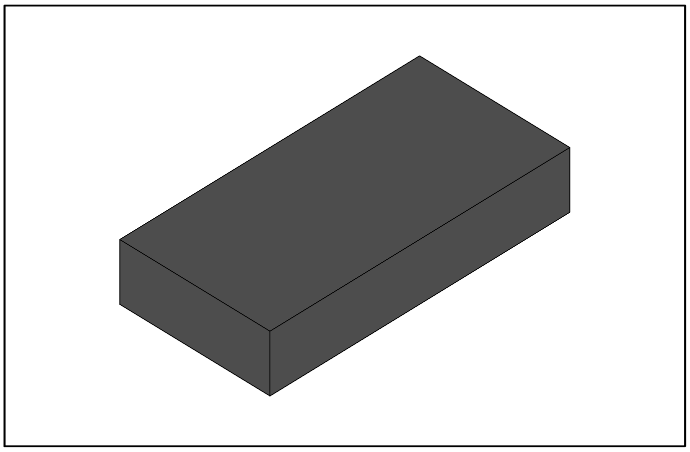
1.4.7 Exercise: The Maple command nextprime(n) will return the next prime after the integer "n". Make a list of pairs of numbers so that the nth element of your list is the pair whose first element is an integer and whose second element is the next prime after "n!". Let the first elements of your lists be the even integers from 8 to 20.

1.5 Laying bricks. Applications of sequences and lists

We can now generate sequences and lists with any command which takes in a single integer and produces an output. It even works with pictures. For instance suppose we want to lay bricks to make (a small part) of a sidewalk,a wall, or the like. First we modify our cube maker from the last worksheet to make only red bricks that a eight units long, four units wide, and two units high. Lets call it "BL" (for bricklayer).

```
BL := proc (shift)
local v1, v2, v3, v4, v5, v6, v7, v8, front, back, left, right,
bottom, top, box;
v1 := [0, 0, 0]+shift; v2 := [8, 0, 0]+shift; v3 := [8, 4, 0]+shift;
v4 := [0, 4, 0]+shift; v5 := [0, 0, 2]+shift; v6 := [8, 0, 2]+shift;
v7 := [8, 4, 2]+shift; v8 := [0, 4, 2]+shift;
front := [v1, v2, v3, v4]; back := [v5, v6, v7, v8];
left := [v4, v8, v5, v1]; right := [v3, v2, v6, v7];
bottom := [v3, v4, v8, v7]; top := [v1, v2, v6, v5];
box := [front, back, left, right, top, bottom];
plots[polygonplot3d](box,color=red,style=patch, scaling=constrained )
end:
```

```
BL([0,0,0]);
```



Suppose we want to make a stack of 10 bricks. Of course we could "personally" make each brick:

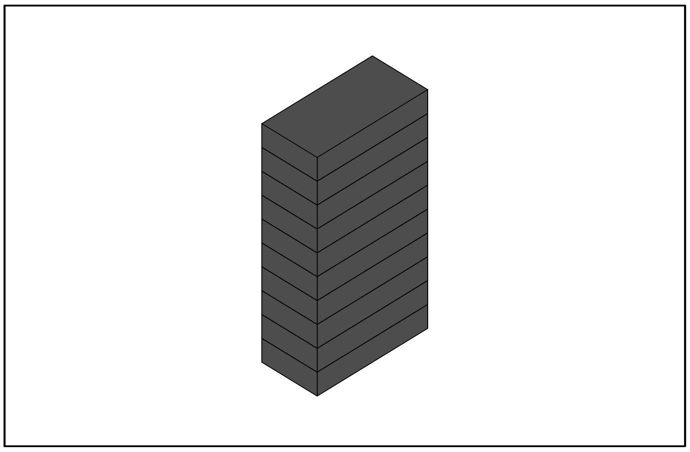
```
b1:=BL([0,0,0]):
b2:=BL([0,0,2]):
b3:=BL([0,0,4]):
b4:=BL([0,0,6]):
b5:=BL([0,0,8]):
b6:=BL([0,0,10]):
b7:=BL([0,0,12]):
b8:=BL([0,0,14]):
b9:=BL([0,0,16]):
b10:=BL([0,0,18]):
```

Then we produce a list to feed to **plots[display]** . Note that we terminate with a semicolon. The reader might want to change it to a colon and execute the command to remind him/herself why - then change it back to a colon and execute again.

```
tenbricks:=[b1,b2,b3,b4,b5,b6,b7,b8,b9,b10]:
```

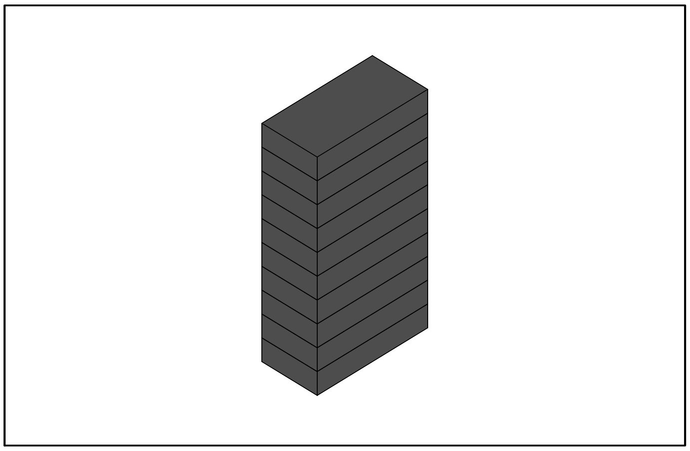
Now we can look at the picture:

```
plots[display](tenbricks);
```



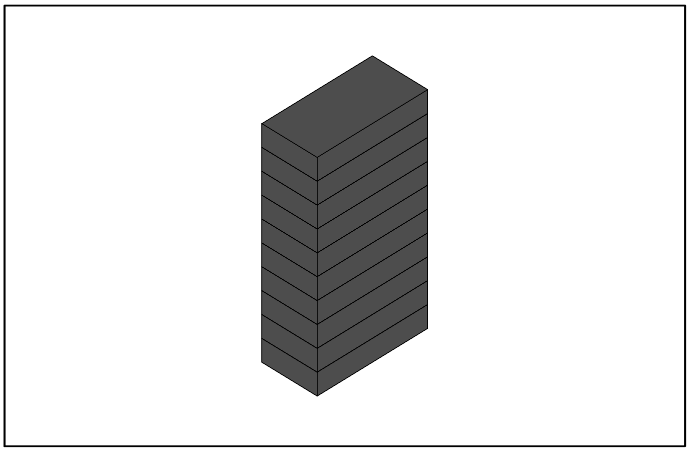
We observe that what we are doing is in making the nth brick is just BL([0,0,2*n]); and this depends only on the integer "n" (we are starting with n=0). So to get our list all we need do is the following. Again we terminate with a colon ":" as we don't want to see the list in this form - we only want the picture.

```
bricklist:=[seq(BL([0,0,2*n]),n=0..9)]:plots[
display](bricklist);
```



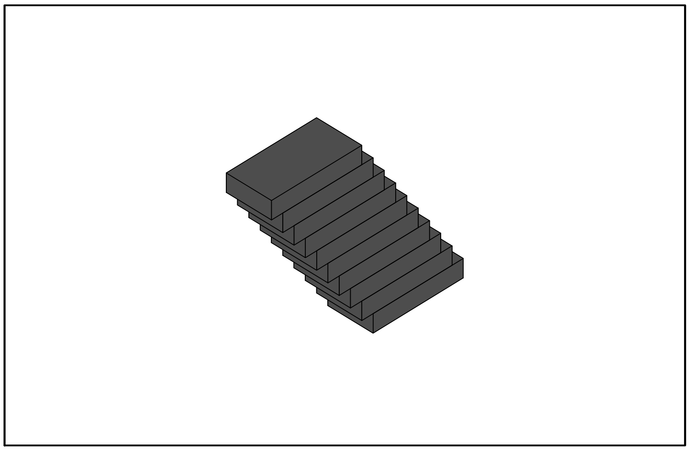
We can improve even further with a compound command which simply combines the previous steps

```
plots[ display ]([ seq(BL([0,0,2*n]), n=0..9)
]);
```



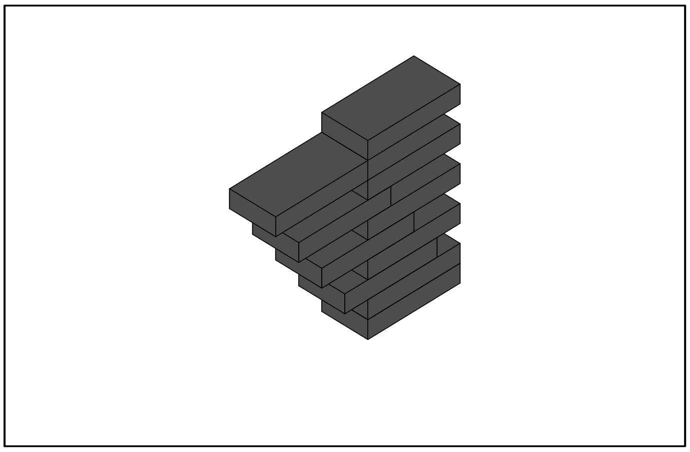
Suppose we want our bricks in a "staircase" pattern, say each brick is to shift over by one unit in the positive "X" direction. Since the description of the nth brick depends solely on "n" we can do it.

```
plots[display] ([seq(BL([n,0,2*n]),
n=0..9)]);
```



Suppose we want only the even numbered bricks to shift

```
plots[display] ([ seq(
BL([n*(1+(-1)^n)/2,0,2*n] ), n=0..9) ]);
```



1.6 EXERCISES: Making Brick Patterns With seq

1.6.1 Exercise: Stack the ten bricks in so that they shift both in the "X" and "Y" directions, making each brick after the first shift 1/3 of a unit in the positive "X" direction and 1/4 of a unit in the negative "Y" direction.

1.6.2 Exercise: Stack the ten bricks so that they don't shift in the "Y" direction but we want each odd numbered brick to be shifted 1 units more than the previous odd numbered brick in the positive "X" direction and the ith even numbered brick to be shifted 1 unit more than the previous even-numbered brick in the negative "X" direction.

1.6.3 Exercise: Using the "seq" command place the ten bricks so that are end to end.

1.6.4 Exercise: Using the "seq" command place the ten bricks so that they are in a rectangular pattern of five rows and two columns.