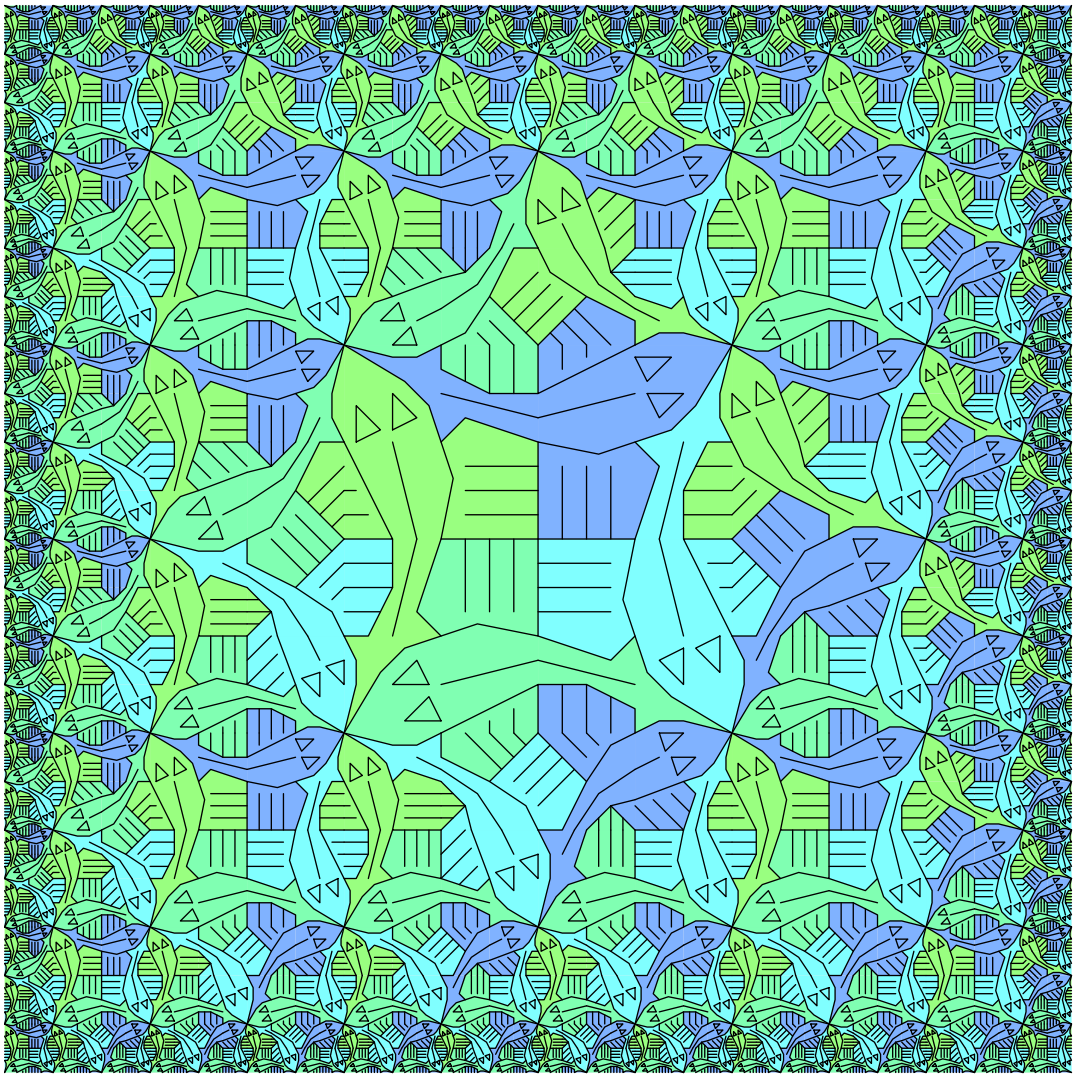


GeomLab Workbook

Michael Spivey
Oxford University Computing Laboratory
http://www.cs.ox.ac.uk/geomlab

Introduction

This booklet will introduce you to some of the most important ideas in computer programming in an interactive, visual way through a guided activity. The activity is about a language for describing pictures. You will use a computer program that lets you type a sentence in the language, and shows you the picture that it describes. At first, the pictures you will make will be simple combinations of stick figures: three men in a row, or one man supporting two smaller men. But gradually, the pictures will get more complicated, and soon we will be describing pictures like the one on the entry page of this website. That picture looks frighteningly complicated, but like all the pictures we will describe, it consists of a few basic tiles combined according to some simple rules.

The point of this activity is not that the language of pictures is useful in itself (although it has important uses in describing the 'pictures' that go into making integrated circuits), but that complex pictures are a metaphor for the complex behaviours that are shown by computer programs. For example, a word processor takes simple actions like printing an individual letter or digit, and combines them into the immensely complex activity of printing an entire document, with each letter and digit in its right place. A spreadsheet takes the simple actions of adding or subtracting or multiplying two numbers, and combines these actions in complex ways to carry out elaborate calculations. The aim of this activity is to let you understand something of what it is like to write a computer program, why it is that computer programming is difficult, and (we hope) why computer programming is a fascinating activity that can be totally absorbing.

To succeed in this activity, you will need to be prepared to think very carefully about what will happen when you write different expressions in our language of pictures. After a while, we will reach sentences that are sufficiently complicated that you will not be able to predict in complete detail what the computer will draw. Part of the point of this activity is that, although the rules for interpreting sentences in the language are simple and known, the effect of applying them systematically can be difficult to imagine. Because you will be using a computer, when we ask you to think about what picture will appear, it will be very easy just to type in the sentence, press the button and see. That's all very well, but you will get a lot more out of the activity if you can explain to yourself why the computer draws the pictures it does.

Computers are very literal creatures, and you will soon discover that, unless you type in each expression *exactly* as you see it written on the worksheet, the computer will very likely complain that it does not understand what you mean, usually by giving a rather

cryptic 'error message'. Any little mistake, such as a spelling error or a missing bracket, will prevent the computer from doing what you want. This is just a fact of life when using computers; people have tried to make computer programs that can correct small errors, but (like the spelling checkers that come with word processors) the 'corrections' are often even less correct than the original mistakes, so that idea is usually more trouble than it is worth. At first, it seems humiliating that the computer is always complaining about 'errors' in your work, but computer programmers soon become used to just correcting the mistake and trying again.

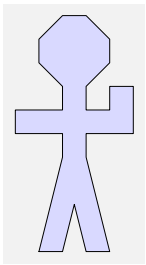
Don't be afraid of trying different expressions to see what the computer will do. The worst thing that can happen is that the computer will not be able to make sense of what you type, and you will get another of those 'error messages'. Or perhaps the computer will draw a picture that is not quite the one you had in mind; in that case, you should try to understand why the picture looks as it does, then try again with a different expression. Just occasionally, the picture that appears will not be the one you wanted, but will be interesting in an unexpected way: that's one of the delights of working with computer graphics.

Worksheet 1: Above and beside

Among other things, GeomLab is a language for describing pictures. The GeomLab program lets you write a description in the left-hand part of the window, and will then show you the corresponding picture in the right-hand part.

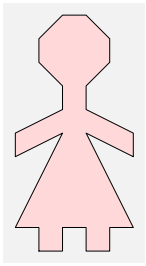
To get you started, we have provided several pre-drawn pictures and given them names. For example, the name `man` stands for a stick-figure picture of a man:

```
man
```



(whenever we show an expression and a picture like this, we mean that GeomLab associates the picture on the right with the description on the left.) Similarly, `woman` stands for a stick-figure woman:

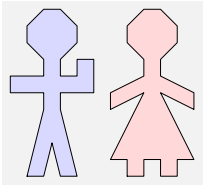
```
woman
```



There are also pictures named `tree` and `star` that you should try out for yourself.

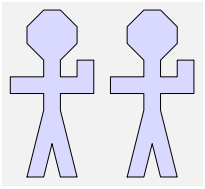
If we have two pictures, then GeomLab lets us put one beside the other using the *operator* `$`. So the expression `man $ woman` stands for the stick-figure man next the the stick-figure woman:

```
man $ woman
```



We can also put one copy of the man picture next to another:

```
man $ man
```

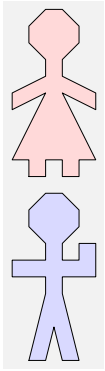


You shouldn't find it hard to guess what result will come from writing `woman $ man`, but (if you have this worksheet on paper) you might like to sketch it in the space below before trying it with the GeomLab program:

```
woman $ man
```

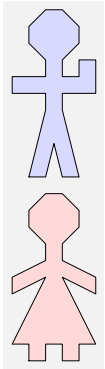
There's another operator, written `&`, that puts one picture above another:

```
woman & man
```

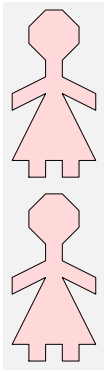


Just as the order of the pictures matters with `$`, so it matters with `&` also, as we can see here:

```
man & woman
```

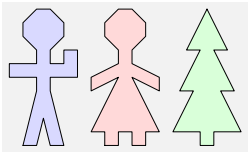


I hope you can see for yourself how to get this next picture:



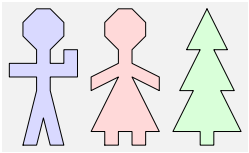
So far, we have used the operators `$` and `&` to combine pictures given by their names, but it is also possible to combine pictures in more than one stage, using the result of one operation as an input to another. For example, we can put a man, a woman and a tree all in a row like this:

```
(man $ woman) $ tree
```



I have used brackets in this expression, just the same way that they are used in an algebraic expression like `(x + y) + z`, to ensure that the man and the woman are combined first, and then the tree is put to their right. As it happens, in this case it doesn't matter, because putting in the brackets the other way gives the same result:

```
man $ (woman $ tree)
```



This is similar to ordinary algebra, where the equation

```
(x + y) + z = x + (y + z)
```

is always true. Use GeomLab to check that you really do get the same picture from both the expressions shown above. Then do an experiment with two expressions that put a man, a tree and a star into a vertical column.

Like addition and multiplication in algebra, the operations `$` and `&` are *associative*, in that the equations

```
(p $ q) $ r = p $ (q $ r)
```

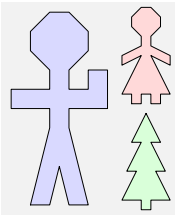
and

```
(p & q) & r = p & (q & r)
```

are always true. As we saw earlier, though, these operations are not *commutative*, because `man $ woman` and `woman $ man` are different pictures.

Now let's try using both `$` and `&` in the same expression. Try this:

```
man $ (woman & tree)
```

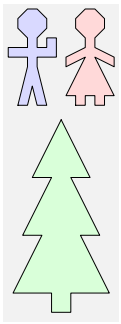


Something new is happening here. When GeomLab combines two pictures using `$`, their sizes are adjusted so that the pictures have the same height, without changing the shape of either picture. Since `woman & tree` is naturally twice the height of the single picture `man`, that picture is shrunk to half size before joining it with the man. (Whatever the size of the resulting picture, GeomLab draws it as large as possible on the computer screen.)

Although this rule seems new, in fact it is consistent with what we have seen already, because the heights of `man` and `woman` already match, and when we form `man $ woman` so do their natural widths. If you look at the pictures `man $ star` and `man & star`, however, you will see that the relative sizes of man and star are different in the two cases, so that the heights match when they are put side-by-side, but it is the widths that are made to match when one is put atop the other.

The next thing to notice is that the brackets do matter in the expression `man $ (woman & tree)`, because `(man $ woman) & tree` is a different picture:

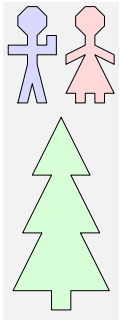
```
(man $ woman) & tree
```



Thinking of algebra again, this fact becomes less surprising. We know that $3 \times (4 + 5)$ is different from $(3 \times 4) + 5$; in fact one of them evaluates to 27 and the other to 17.

In algebra, there are conventions about which of these expressions is meant if we write $3 \times 4 + 5$ without brackets. The $\times$ operation is said to *bind more tightly* or have a *higher priority* than the $+$, and that obliges us to read the expression as if it were $(3 \times 4) + 5 = 17$. Similarly, there is a rule in GeomLab that `$` binds more tightly than `&`, and that allows us to predict the result when we leave out the brackets:

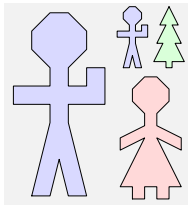
```
man $ woman & tree
```



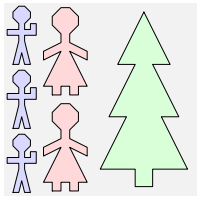
Try it!

Although this rule allows us to predict what will happen when brackets are left out, we are still free to put them in when we want to force another interpretation, like this:

```
man $ (man $ tree & woman)
```

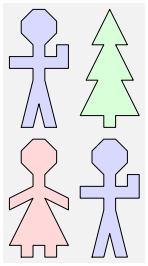


Can you work out what expression gives this picture?



Now look at this picture:

```
(man $ tree) & (woman $ man)
(man & woman) $ (tree & man)
```



Both expressions give the same picture (try it!), so these two expressions are equivalent.

If the pictures `p`, `q`, `r` and `s` all have the same shape (as they do here), then the two expressions

```
(p $ q) & (r $ s)
```

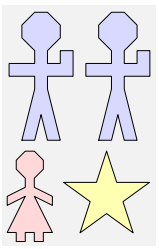
and

```
(p & r) $ (q & s)
```

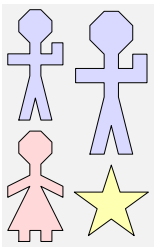
produce the same picture.

But if the component pictures have different shapes, this is not always true. For example, applying the rule to `((man $ man) & (woman $ star)` we get two different pictures:

```
(man $ man) & (woman $ star)
```



```
(man & woman) $ (man & star)
```



We can see that the two pictures are not the same, since the sizes of some of the figures have come out differently. If you are good at algebra, you might like to try to work out exactly when the equation

```
(p $ q) & (r $ s) = (p & r) $ (q & s)
```

is true for four pictures `p`, `q`, `r` and `s`. A hint: it has to do with the ratios between the widths and the heights of the pictures. The scaling rule of GeomLab means that these ratios do not change when the pictures are scaled up or down; both sides of the equation will give the same result if the boundaries between the pictures meet in a point.

In this sheet, we have seen that it makes to work with formulas whose values are not numbers but pictures. We can have constants that stand for fixed pictures, and we can have operators that combine pictures to make new ones. Just as in ordinary algebra, it makes sense to think about the relative priority of the operators: just as convention dictates that multiplication is done before addition where they appear together in a formula, we can make the convention that `$` is done before `&`.

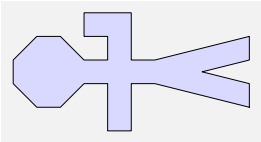
In ordinary algebra, some equations are true no matter what values we put for the variables they contain: for example, the equation `a + (b + c) = (a + b) + c` expresses the fact that addition is associative. Similarly, we can look for equations that always hold between formulas in our new kind of algebra.

Looking beyond this little world of pictures, there are many places in programming and computer science where the algebraic idea of having a set of values with operations on them is relevant. For example, tables of information in databases can be combined with algebraic operations that match the values in a column of one table with values that appear in a different column of another table. This is a good way of organising a database system so that people who use it in their programs are insulated from the way the data is stored.

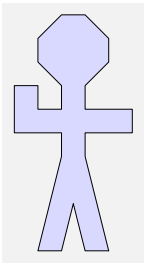
Worksheet 2: Rotations and reflections

Another two functions that can be used on (or *applied to*) pictures are `rot` and `flip`. The function `rot` stands for *rotate*, and twists a picture anticlockwise by 90 degrees. The function `flip` reflects a picture about a vertical axis, so it looks like the picture has been flipped over:

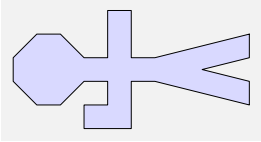
`rot(man)`



`flip(man)`



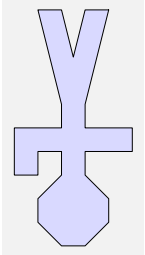
`rot(flip(man))`



Look carefully at the first and third pictures above – they are *not* the same!

Since `rot` and `flip` are functions, they can be applied several times to an argument. For example, `rot(rot(man))` produces the following image:

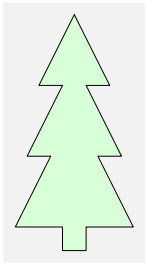
`rot(rot(man))`



Similarly, `flip(flip(man))` is allowed, although this will simply produce the original picture of `man`. Try it out for yourself in GeomLab. What will `rot(rot(rot(tree)))` look like? Sketch the image in the space below:

```
rot(rot(rot(tree)))
```

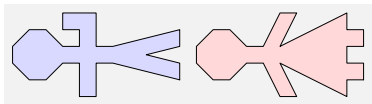
Check your answer in GeomLab. Since the function `rot` rotates a picture, it should be possible to get back to the original picture by using `rot` several times. Fill in the expression missing in the space below – you must use the function `rot` at least once, so don't just write "`tree`"!



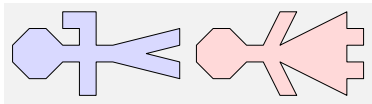
Check your answer in GeomLab.

So far we've used rot and flip on just *constant* pictures like `man` and `tree`. Now let's try the expression `rot(man & woman)`:

```
rot(man & woman)
```



Using the picture as a guide, how can this expression be rewritten using only the functions `rot` and `$`? Fill in the expression below:



Check your answer in GeomLab.

Generally it's true that

```
rot(p & q) = rot(p) $ rot(q).
```

Is it also true that

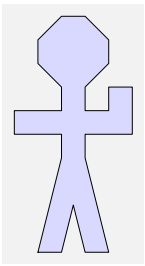
```
rot(p $ q) = rot(p) & rot(q)?
```

Try it out on some examples in GeomLab, and see if the images produced are the same.

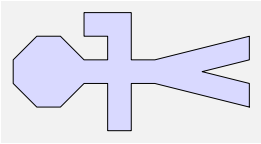
If the two expressions are *not* the same, suggest an equation that *is* true:

We can create a variety of different images using the functions rot and flip on the constant picture `man`:

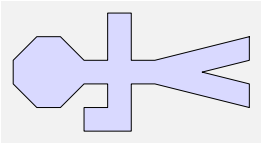
`man`



`rot(man)`

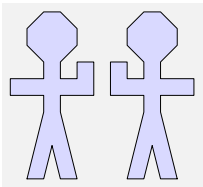


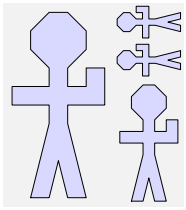
`rot(flip(man))`

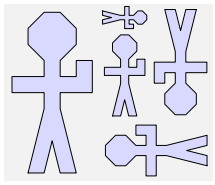


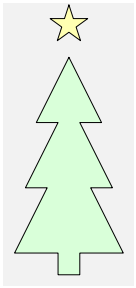
How many different pictures are there? How many pictures can we create using the functions `rot` and `flip`? IfBook—Write your answer with an explanation in the space below:

Now try to find expressions that result in these pictures:









For this last picture, you will need the constant picture `star`, and it may help you to make the star be the right size if you know that `blank` is a square picture that is entirely blank.

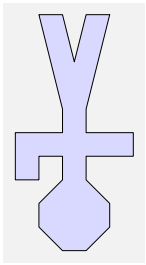
In this sheet, we have added more operations to our algebraic language of pictures, so that we can now rotate and reflect pictures as well as putting them side-by-side or one above another. In addition to adding more operations, we have also found new algebraic identities that relate the operations to each other. Some of these identities make it possible to move instances of `rot` and `flip` inwards in any expression, so that it becomes a combination (using `$` and `&` of rotated or reflected primitive tiles. This is more-or-less what the computer does in order to draw the pictures your program creates.

In a wider computer science setting, similar algebraic identities are used internally by compilers, the programs that translate high-level programs written by human programmers into the low-level instructions that a machine can follow step by step. The compiler can use algebra to simplify the low-level program it creates, for example by deleting two operations if they cancel each other out. This makes the low-level program smaller to store and faster to obey.

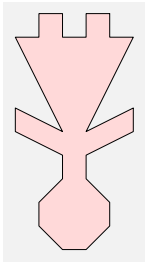
Worksheet 3: Definitions and functions

Remember that `rot` can be applied twice to a picture to turn it upside down:

```
rot(rot(man))
```



```
rot(rot(woman))
```



It would be convenient to make a new function that could be used when we want two successive rotations to be applied to a picture, as in the examples above. We can do this using an expression with `define`, like this:

```
define rot2(p) = rot(rot(p))
```

The command `define` makes the expression that follows it on the left hand side of the = sign to stand for the expression on the right of the = sign.

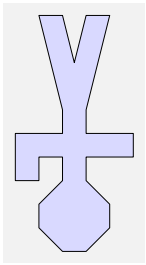
When you type this definition in place of an expression, GeomLab responds with the message

```
--- rot2 = <function>
```

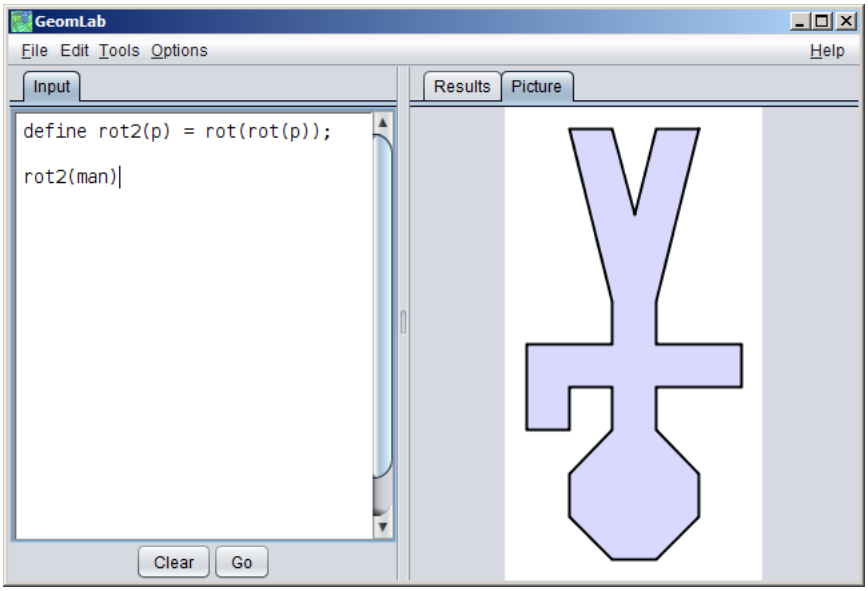
This shows the `rot2` has been defined as a *function*: it is not a picture in itself, but it produces a picture when we supply an argument (i.e., the picture that is to be rotated).

Having defined the function `rot2`, we can use it in the same way as any other function:

```
rot2(man)
```



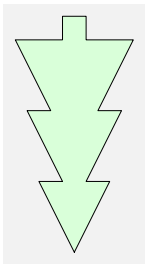
In order to use a defined function in an expression, we just type both the definition and the expression in the input window, like this:



Each definition ends with a semicolon, and that helps GeomLab to keep it separate from the following definition or expression.

Replacing `man` by `tree` gives another upside down picture:

```
rot2(tree)
```



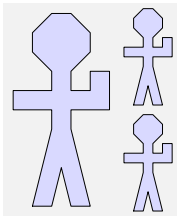
Note that the variable `p` used in the expression defining `rot2` is a *placeholder*. This means that the function `rot2` can be applied to any expression put in place of variable `p`, such as `man` or `woman`.

Let us consider another function definition:

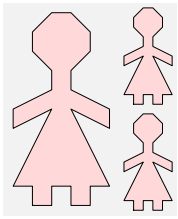
```
define f(p) = p $ (p & p)
```

This makes a picture that contains three copies of the argument picture:

```
f(man)
```



```
f(woman)
```



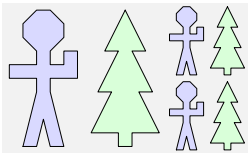
What will the image produced by `f(man) $ tree` look like? Sketch the picture here:

```
f(man) $ tree
```

Check your answer in GeomLab: again, you must enter simultaneously both the definition of `f` and the expression that uses it.

Here is the image produced by `f(man $ tree)`:

```
f(man $ tree)
```

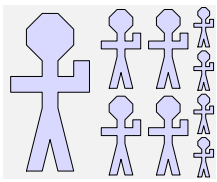


This image is different from the one produced by `f(man) $ tree`. This is because the brackets in the first example are not around `tree`, so the function `f` is applied only to `man`, whereas the second example has brackets that include `tree`, so the function `f` is applied to the expression `man $ tree`.

```
f(man $ tree) = (man $ tree) $ ((man $ tree) & (man $ tree))
f(man) $ tree = (man $ (man & man)) $ tree
```

Now look at the picture `f(f(man))`:

```
f(f(man))
```



Looking at the picture above, there are *nine* men produced by applying function `f` twice to `man`. What happens if we apply function `f` *three* times to `man`, as in `f(f(f(man)))`? How many men do you think will be in the picture? Try it!

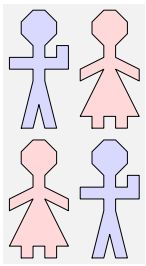
As described in Worksheet 2, a function takes one or more *arguments*. For example, the function `f` that we defined earlier takes one argument, which could be `man`, `woman` or any other picture expression.

Let us define a function g that takes two arguments:

```
define g(p, q) = (p & q) $ (q & p)
```

Here is what `g(man,woman)` looks like:

```
g(man, woman)
```



What will happen if we type `g(man,g(man,woman))` into GeomLab? Sketch in the space below the image that you think will be produced:

```
g(man, g(man, woman))
```

Check your answer in GeomLab.

In this sheet, we have seen how to introduce new functions into the GeomLab language by defining them with a formula that gives their value. This does not increase the variety of pictures that we can describe, for we can always eliminate the new functions from any formula by substituting the defining formula of the function in each place it is used. In essence, this is what the computer does when we ask it to evaluate a formula containing functions we have defined. Although defining functions does not give us new pictures in principle, nevertheless in practice it does let us describe complex pictures much more easily.

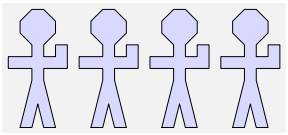
In a wider programming context, defining new functions is one of the chief ways programmers keep the complexity of computer systems under control. A good function does a well-defined job, such as solving an equation or displaying a web page, and does it in such a way that you don't need to understand how the function works in order to use it – just as you don't need to understand how a CD player works in order to play music on it. In this way, programmers constantly enrich the vocabulary of their language, making it possible to give succinct instructions for a wider and wider range of tasks. One of the hallmarks of an experienced programmer is an ability to simplify programs and make them easier to understand by introducing appropriate functions.

Worksheet 4: Recurrences and recursion

Suppose we want to draw rows of men of different lengths. It would be useful to have a function `manrow(n)` that, for any value of `n`, would give a row of `n` men. How can we define this function so that it works for any value of `n` .

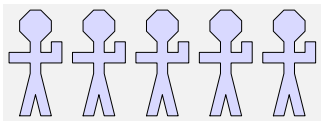
For example `manrow(4)` would be a row of 4 men; let's call that picture `r4`:

```
define r4 {{=}} man $ man $ man $ man
```



Similarly, let us call a row of five men `r5`:

```
define r5 {{=}}
~~man $ man $ man $ man $ man
```



What relates `r4` and `r5`? How can we go from a row of four men to a row of five men? We just need to add another man! So we can say:

`r5 = r4 $ man`.

Similarly, we can say that `r4 = r3 $ man` if `r3` is a row of three men.

Writing your answers in the same way, work out equations for `r3` and `r2`. IfBook—

The expression `r1` refers to a row of just one man, which is the same as the expression `man`. Hence we should write the equation

`r1 = man`.

It would be ideal if we could define a function `manrow` that can take a number as an argument, and produce a row of men that is as long as that number. The way to do this is tricky to understand, so look at the definition below, and then read the explanation that follows it:

```
define manrow(n) = manrow(n-1) $ man when n > 1
  | manrow(1) = man;
```

This definition of the function `manrow` consists of two equations, separated by a vertical bar character `|`. The first equation is used when `n > 1`, and states in general form our observations that `r4 = r3 $ man`, and `r3 = r2 $ man`, and so on. The other equation is simpler, and just restates the fact that `r1 = man` in terms of the function `manrow`.

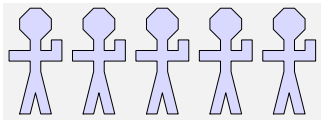
You should enter this definition into GeomLab. On a British PC keyboard, the symbol `|` is entered by holding down the `Shift` key and pressing the `\\` key, just to the left of the `Z`. At first sight, this equation appears to be defining the function `manrow` in terms of itself, and because of this, we call it a *recursive* definition. Despite the self-reference, the definition works because it shows us how to calculate, say, `manrow(5)` assuming we already know how to calculate `manrow(4)`, and applying the equation repeatedly can bring us down to the base case `manrow(1) = man`.

Having defined `manrow` as aboive, if you type an expression like manrow(5), then GeomLab expands the definition like this:

```
manrow(5) = manrow(4) $ man
  = manrow(3) $ man $ man
  = manrow(2) $ man $ man $ man
  = manrow(1) $ man $ man $ man $ man
  = man $ man $ man $ man $ man.
```

The result is the image we expected:

```
manrow(5)
```



What does `manrow(4) & manrow(3) & manrow(2)` look like? Sketch the picture here:

```
manrow(4) & manrow(3) & manrow(2)
```

This picture looks a bit like a crowd, so let's try to define a function that can create crowds of different sizes.

To get a realistic-looking crowd, we will need to have rows that get larger in the number of men (and smaller in height) as we travel from bottom to top of the picture. The function `crowd(m, n)` can be defined so that `m` is the length of the bottom row in the picture, and `n` is the length of the top row. To save ourselves work, we can use the previously defined function `manrow` to draw the rows that we want.

If `m` and `n` are the same, then the crowd will consist of just one row:

```
crowd(m, m) = manrow(m)
```

Otherwise, we are considering an expression of the form `crowd(m, n)`, where `m < n`. This crowd consists of a top row containing `n` men, and below it a smaller crowd with rows ranging from `m` to `n-1` men:

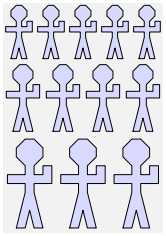
```
crowd(m, n) = manrow(n) & crowd(m, n-1)
```

We can put these two equations together to make a recursive definition of the function `crowd`:

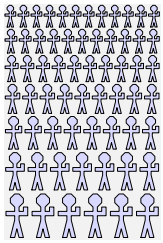
```
define crowd(m, n) = manrow(n) & crowd(m, n-1) when m < n
  | crowd(m, m) = manrow(m)
```

Here are two example images produced using our `crowd` function:

`crowd(3, 5)`



`crowd(6, 12)`



Try it, by putting the definitions of both `manrow` and `crowd` together one with one of these expressions, all in the same text.

This sheet has introduced an important new idea: that we can take a recurrence relation that describes a sequence of pictures, and turn it into the definition of a recursive function that generates pictures in the sequence. Simple recursive definitions have two parts: a rule that generates an element of the sequence from the previous element, and a starting rule that defines the first element. The idea of defining functions recursively is important, because it opens the possibility that a finite program can generate an infinite variety of behaviour by varying the argument that is passed to the function.

With a non-recursive function, each time the function is referred to, the formula that is the body of the function is used just once. With a recursive function, substituting the body of the function for a use of it may leave a formula that still contains a use of the function, so the definition may be used many times in computing the value of the function. In a wider setting, it is recursion that lets us use functions to model, understand, and compute with complex processes.

Worksheet 5: Spirals and zig-zags

This worksheet uses two constants `straight` and `bend` that look like square tiles:

```
straight
```

```
bend
```

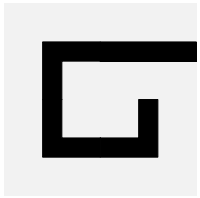
Let's introduce some short names for rotated versions of these basic shapes:

```
define s = straight
define s1 = rot(s)
define b = bend
define b1 = rot(b)
define b2 = rot(b1)
define b3 = rot(b2)
```

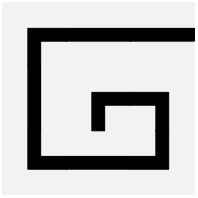
We can now draw various shapes by putting together the tiles that we have just defined:

```
define spi1 {{=}} b
```

```
define spi2 {{=}} (b $ s) & (b1 $ b2)
```

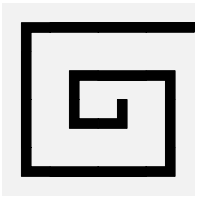


```
define spi3 {{=}}
~~(b $ s $ s) & (s1 $ b $ b3) & (b1 $ s $ b2)
```



The picture for `spi4` is shown below on the right. Define `spi4` yourself in the space provided, then use GeomLab to test your expression, and see if you are right:

```
define spi4 {{=}}
```



How can we draw more spirals with more and more turns without typing more and more complicated expressions? The answer is to write a program to do it.

We can start by working out how to draw one *arm* of a spiral. This is a number of `straight`'s, with a `bend` at the end. As we did with the rows of men in Worksheet 4, let's begin by drawing the first few sides one at a time, and then look for a pattern:

```
define a1 {{=}} bend
```

```
define a2 {{=}} bend $ straight
```

```
define a3 {{=}} bend $ straight $ straight
```

From the examples above, we can see that we start with a `bend` and add `straight`'s one at a time. We can express this process using a recursive function, as described in Worksheet 4:

```
define arm(n) = arm(n-1) $ straight when n > 1
  | arm(1) = bend
```

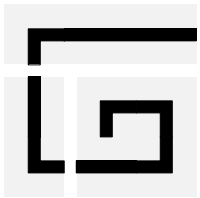
The new function `arm` can now be used to draw sides of desired length:

```
arm(3)
```

```
arm(4)
```

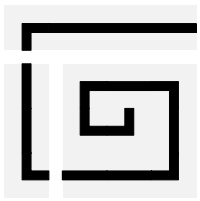
Let's now try to use our function `arm` to create some spirals. How can we get from `spi2` (defined earlier) to `spi3`, and from there to the next spiral, and so on? We can see that `spi1` is a 2×2 arrangement of tiles, and the same tiles appear as one corner of `spi3`, after being rotated by 180 degrees. The rest of `spi3` consists of a rotated copy of `arm(2)` and a copy of `arm(3)`, as shown in the image below:

```
arm(3) & (rot(arm(2)) $ rot2(spi2))
```



(I've separated the parts of the picture a bit, so that you can see how it fits together). The resulting image looks exactly like `spi3`. This process can be repeated using `spi3` to produce `spi4`, as shown below:

```
arm(4) & (rot(arm(3)) $ rot2(spi3))
```



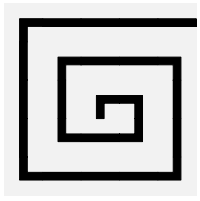
We can see that this process can be applied to any spiral, so it must be possible to use it in a recursive function that draws spirals of any size. Since we have defined `spi1` as simply `bend`, we can use this as a base case for the following recursive definition:

```
define spiral(n) = ???? when n > 1
  | spiral(1) = bend
```

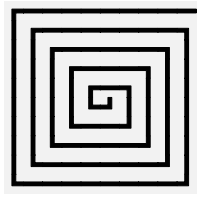
Fill in the missing expression in the definition above. Check your answer in GeomLab (type in the completed definition, and try a few examples to see whether the correct picture is produced).

With the completed definition, we can draw spirals with as many turns as we like:

```
spiral(5)
```



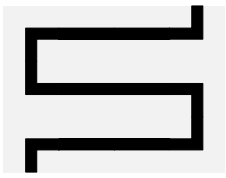
```
spiral(10)
```



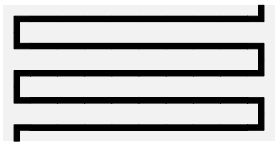
(**Note:** When a complex spiral is drawn, the turns will get very small, and you'll start to see them merge together in a way that may look a bit uneven on the computer screen. Don't worry, this is normal!)

Now here is a challenge for you: define a function `zigzag` that can draw zig-zags like these:

```
zigzag(4,3)
```



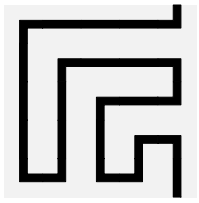
```
zigzag(10,5)
```



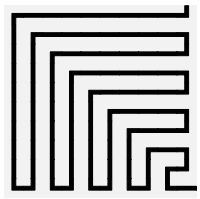
The picture `zigzag(m,n)` should be `m` units wide and `n` units high.

Also, define a function `zagzig` that can draw shapes like these:

```
zagzig(5)
```



```
zagzig(10)
```



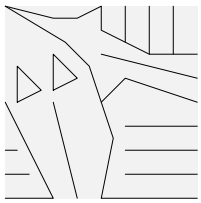
In this sheet, we have seen more examples of recursive functions that generate a sequence of patterns, each a little more complex than the one before. Each time, the key to finding the proper definition of the function is to work out exactly how a copy of each member of the sequence is embedded in the next member.

Worksheet 6: Escher pictures

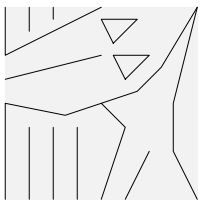
Let's see how to draw the Escher picture "Square Limit" that is on the front cover of this workbook. What is fascinating about this picture is that the fish get smaller and smaller as they get closer the the edge of the picture, making it possible in principle to draw an infinite number of fish in a finite space. Instead of drawing this infinitely complex picture, we will instead define a function that can produce any finite portion of it. At first, we'll aim at making a black-and-white version of the picture.

The picture is made from four square tiles: `A`, `B`, `C` and `D`:

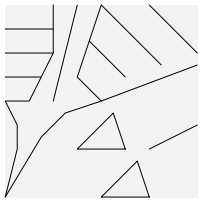
`A`



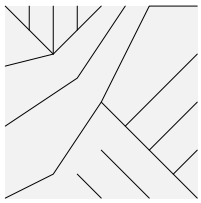
`B`



`C`

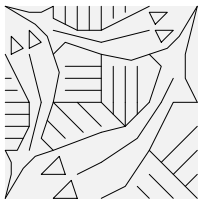


`D`



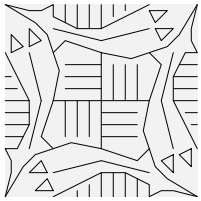
These tiles fit together to make a larger tile that we'll call `T`:

```
define T {{=}} (A $ B) & (C $ D)
```



It's also possible to fit together four rotated copies of tile `A` to make another tile that we'll call `U`:

```
define U {{=}} (A $ rot3(A)) & (rot(A) $ rot2(A))
```



As you can see, tile `U` forms the centre of the Escher picture.

The important thing about tile `T` is that it fits next to a smaller copy of itself, as is shown here:

```
(T & blank) $ T
```

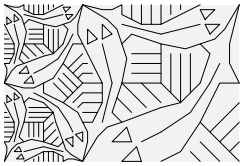
In this picture, the blank square forces the copy of `T` shown on the left to be half the size of the one on the right, but the tiles still fit together nicely.

It's also possible to fit a small, rotated copy of `T` next to `T`, like this:

```
(blank & rot(T)) $ T
```

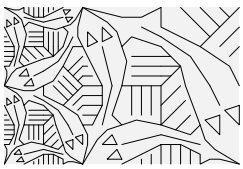
In fact, all three copies of `T` can be fitted together at once:

```
define p {{=}} (T & rot(T)) $ T
```



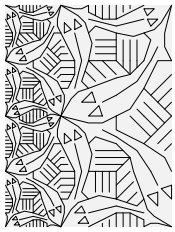
What's more, the same trick can be played with a rotated copy of `T` on the right:

```
define q {{=}} (T & rot(T)) $ rot(T)
```



And the pictures `p` and `q` fit together vertically:

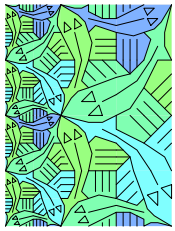
```
p & q
```



It's a remarkable aspect of Escher's genius that he was able to design a tile that works like this.

Now is a good time to introduce a trick that allows us to get colour pictures. GeomLab has been set up so that it can colour in any picture made up from the tiles `A`, `B`, `C` and `D`, choosing the colour of each fish according to the direction it is pointing. Try typing

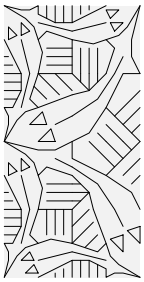
```
colour(p & q)
```



to see the previous picture coloured in this way. This picture looks similar to the part of the Escher picture that is close to the left-hand edge. This trick can be played with any of the pictures that are made up of tiles `A`, `B`, `C` and `D` – and the colours help to make it obvious when the tiles don't fit together properly.

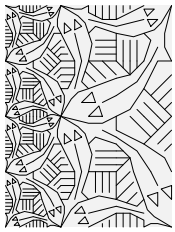
In fact, we can define a sequence of more and more elaborate edge pieces like this: first, `side1` is simply two copies of `T`, one of them rotated:

```
define side1 {{=}} T & rot(T)
```



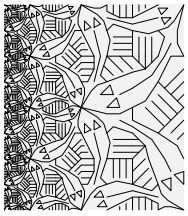
Then `side2` is obtained by joining two copies of `side1` and two copies of `T`, again with one rotated:

```
define side2 {{=}} (side1 $ T) & (side1 $ rot(T))
```



This is the same as the picture `p & q` we made earlier. The next picture in the sequence, `side3`, is obtained in a similar way from `side2`:

```
define side3 {{=}} (side2 $ T) & (side2 $ rot(T))
```

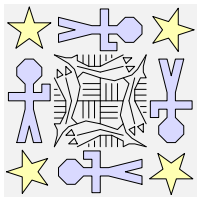


To start putting together the Escher picture from these pieces, we can use a function called `frame`, defined as follows:

```
define frame(c, s, p) =
  (c $ rot3(s) $ rot3(c))
  & (s $ p $ rot2(s))
  & (rot(c) $ rot(s) $ rot2(c))
```

(To save you from typing it out, I've included this definition as part of GeomLab itself). The idea is that `frame({\it c\/}, {\it s\/}, {\it p\/})` is a picture that has the picture `{\it p\/}` in the middle, rotated copies of the picture `{\it s\/}` at the sides, and rotated copies of the picture `{\it c\/}` in the corners. For example, we could frame the picture `U` with copies of `man` and `star`:

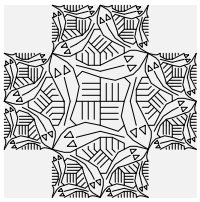
```
frame(star, man, U)
```



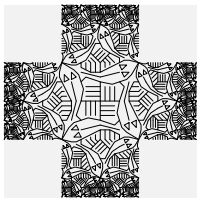
For `frame` to work well, the pictures `{\it c\/}` and `{\it p\/}` should be square, but `{\it s\/}` can be rectangular.

We can use `frame` to put together Escher pictures like this, leaving the corners blank for now:

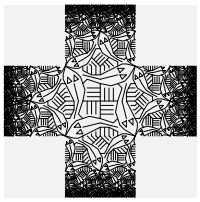
```
frame(blank, side1, U)
```



```
frame(blank, side2, U)
```

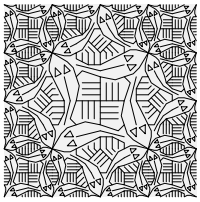


```
frame(blank, side3, U)
```



So all that remains is to work out what would fill in the gaps at the corners. For the first picture, copies of `U` will do:

```
frame(U, side1, U)
```

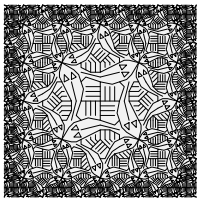


To fit with `side2` or `side3`, something more complicated is needed. If we define

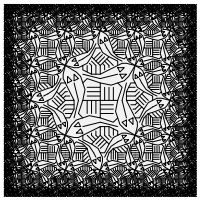
```
define corner1 = U
define corner2 = (corner1 $ rot3(side1)) & (side1 $ U)
define corner3 = (corner2 $ rot3(side2)) & (side2 $ U)
```

then these fit perfectly to make the next pictures in Escher's sequence:

```
define limit2 {{=}} frame(corner2, side2, U)
```



```
define limit3 {{=}} frame(corner3, side3, U)
```



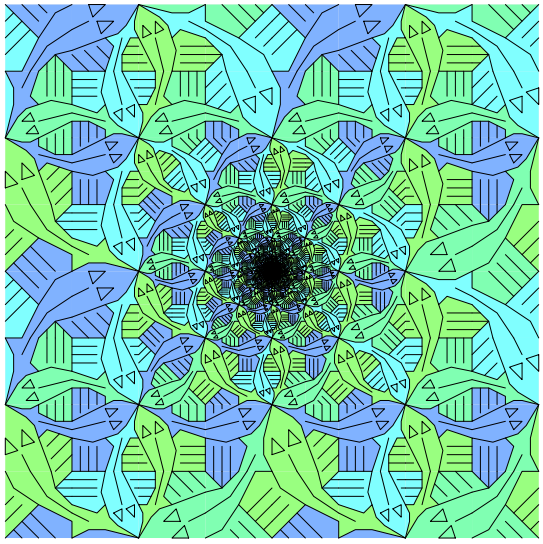
You can make colour versions of these pictures by typing (for example) `colour(limit3)`.

Now two problems for you: first, to define recursive functions called `side({\it n\/})` and `corner({\it n\/})` so that the function `limit({\it n\/})`, defined by

```
define limit(n) = frame(corner(n), side(n), U)
```

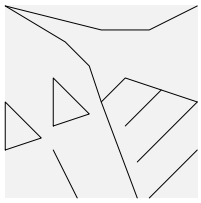
can generate the Escher picture to any desired degree of complexity.

Second: work out how to draw the picture that is shown on the back cover of this workbook, in which the fish get smaller towards the centre. IfWiki—

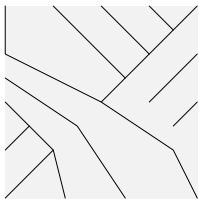


To do so, you will need two more tiles, `E` and `F`:

```
E
```

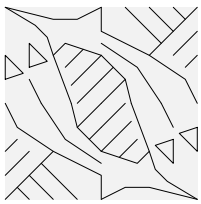


```
F
```



Copies of these two tiles fit together to make a pair of fish chasing each other's tails:

```
define V {{=}} (E $ F) & (rot2(F) $ rot2(E))
```



These fish appear along the diagonals of the picture. You should aim to define an 'inverse limit' function `invlimit({\it n\/})` that can be used to produce any desired degree of approximation to the picture, replacing the center part with `blank`.

Worksheet 7: Space-filling curves

Hilbert curve

By now, I hope you will relish the challenge of working out how to draw pictures without much help from me. If so, here are two families of pictures that are made up from square tiles.

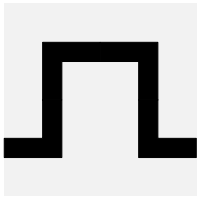
The first family was discovered by the German mathematician David Hilbert (1862–1943), and can be made with the `bend` and `straight` tiles that we used earlier:

`bend`

`straight`

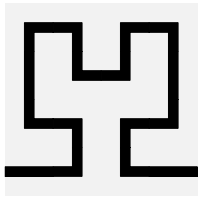
The first curve in the family (let's call it `hilbert(1)`) consists of four copies of `bend`, rotated and joined together:

`hilbert(1)`



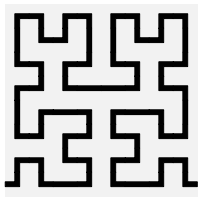
The next curve is a little more complicated:

```
hilbert(2)
```



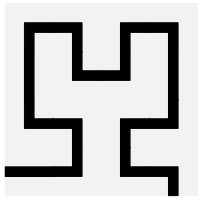
With the third curve in the sequence, a pattern starts to emerge:

```
hilbert(3)
```



It looks as if this curve is made up of four copies of `hilbert(2)` rotated and joined in the right way. But if you look closely, you will see that in each copy of the curve, one of the ends has been bent so as to make it join with the next part of the curve, so that `hilbert(3)` actually consists of four copies of different picture – let's call it `hilbend(2)`:

```
hilbend(2)
```



To draw the Hilbert curves properly, you will need to define *two* recursive functions that depend on each other. You can begin to work out their definitions by asking what `hilbend(1)` would need to look like if it had the same relationship with `hilbert(2)` that `hilbend(2)` has with `hilbert(3)`.

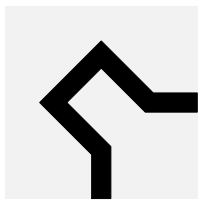
What should `hilbend(3)` look like, and can it be made from copies of `hilbert(2)` and `hilbend(2)` put together in an appropriate way? Does the sequence have to start with `hilbert(1)`? Is there an appropriate definition of `hilbert(0)` that fits the pattern?

These hints should be enough for you to define the two functions for yourself. When you have done so, what happens if you evaluate `hilbert(10)`? Does this justify the term *space-filling curve* that is used to describe the behaviour of curves in this family?

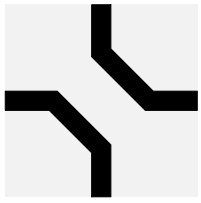
Sierpinski curve

Another family of space-filling curves was discovered by the Polish mathematician Waclaw Sierpinski (1882–1969). This can be drawn using two new tiles that we shall call `nub` and `link`:

```
nub
```

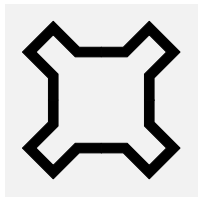


```
link
```



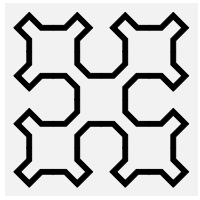
We can make the first curve in the family using four copies of `nub`:

```
sierp(1)
```



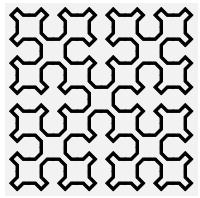
The next member of the family is almost but not quite four copies of this picture joined together:

```
sierp(2)
```



The third member is similarly obtained from four pictures that are not quite the same as `sierp(2)`:

```
sierp(3)
```



Can you write a recursive definition of the function `sierp`, perhaps by defining it together with another function, so that they are mutually dependant?

These two families of curves are called space-filling because (in a precise sense) the curves come close to every point in the square that encloses them. By using some results about continuous functions that are proved in university-level maths, it is possible to use these curves to show a seeming paradox, that there is a continuous function that maps the unit interval to the whole of the unit square.

From a programming point of view, the interesting thing is that – like the Escher picture – these curves are made up of several pieces, each similar to the curve itself.

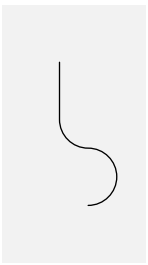
Worksheet 8: Turtle graphics

This worksheet introduces a different way of drawing pictures that will allow us to escape from the world where every non-trivial picture consists of two other pictures put either one *beside* the other or one *above* the other. Instead of joining together tiles, we will give instructions to an imaginary robot called a *turtle*, telling it whether to turn to the right or left, or go straight on. We will write programs that produce the instructions for the robot as a list `y`, then use a function `turtle(y)` that is provided by GeomLab: this function takes the list of instructions and produces a picture of the track that the robot follows.

The turtle obeys a number of different commands, and among them are `ahead(x)`, an instruction to move ahead by `x` units; `left(a)`, an instruction to turn to the left through an angle of `a` degrees, and `right(a)`, a similar instruction to turn to the right. When the robot turns, it moves along the arc of a circle that has diameter 1 unit.

The `turtle` function takes instructions and produces a picture. Its input is not just a single instruction, but a *list* of instructions that are followed one after another. Lists in the GeomLab language are written with square brackets, so that `[left(180), right(90), ahead(1)]` is a list of instructions, and supplying it to the `turtle` function gives the corresponding picture:

```
turtle([left(180), right(90), ahead(1)])
```



The turtle that follows the list of commands always starts pointing to the right; the starting position is in the middle of the bottom of the picture here. The commands say to turn to the left through a half-circle, then to the right through 90 degrees, then to go straight ahead for 1 unit, and the picture is what results from following these instructions.

GeomLab follows the motion of the turtle as it draws the picture, and makes the picture as large as will fit in the window. Depending on how the turtle moves, the starting and ending positions can be anywhere within the window or at its edge.

GeomLab provides a number of operations that act on lists. If `xs` and `ys` are lists, then `xs ++ ys` is a list that contains all the elements of `xs`, followed by all the elements of `ys`. Here's an example that uses lists of numbers instead of lists of commands:

```
[1, 2, 3] ++ [4, 5] = [1, 2, 3, 4, 5]
```

Another operation that is provided is `reverse`, which gives a list that contains the same elements as its argument, but in reverse order:

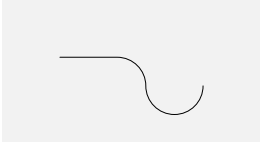
```
reverse([1, 2, 3]) = [3, 2, 1]
```

If `xs` is a list of *commands*, then `opposite(xs)` is a list that is modified so that each command `left(a)` is replaced by `right(a)`, and vice versa:

```
opposite([left(90), right(90), left(180), ahead(1)])
  = [right(90), left(90), right(180), ahead(1)]
```

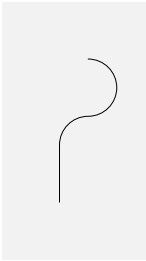
Here is an example that shows the effect of reversing a list of commands:

```
turtle(reverse([left(180), right(90), ahead(1)]))
```



As you can see, the picture us different from the previous one. The function `opposite` gives a picture that is different again:

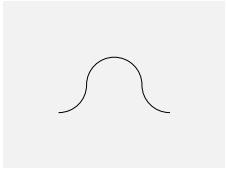
```
turtle(opposite([left(180), right(90), ahead(1)]))
```



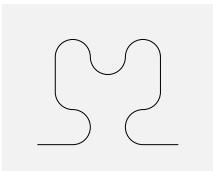
What happens if we apply first `reverse` and then `opposite` to a list of commands?

We can use `turtle` to replicate the Hilbert curves that we drew in Worksheet 7, but describing them now in terms of the turns to left and right that the turtle must make as it follows the curve. The best way to do this is to define a function `hilb(n)` that produces the list of commands that must be obeyed in drawing the `n`'th curve in the sequence. Here are the first three curves, drawn in this way:

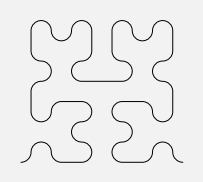
```
turtle(hilb(1))
```



```
turtle(hilb(2))
```



```
turtle(hilb(3))
```



GeomLab draws these curves with smooth turns rather than sharp bends, giving a different effect from the tiled pictures we drew earlier.

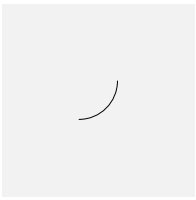
To reproduce these results yourself, you will need to define the function `hilb` that generates the list of commands. For example,

```
hilb(2) = [ahead(1), left(90), left(90), right(90), ahead(1),
  right(90), right(90), left(90), left(90), right(90),
  right(90), ahead(1), right(90), left(90), left(90), ahead(1)]
```

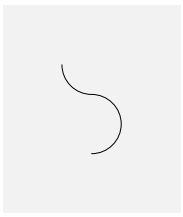
Each of the curves starts in the bottom left-hand corner of the picture, because the turtle can then start off pointing to the right.

The first six *dragon curves* look like this:

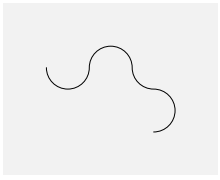
```
turtle(dragon(1))
```



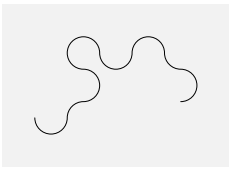
```
turtle(dragon(2))
```



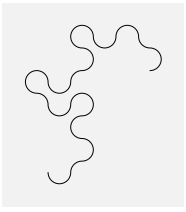
```
turtle(dragon(3))
```



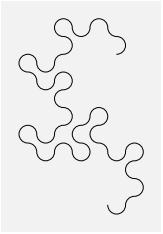
```
turtle(dragon(4))
```



```
turtle(dragon(5))
```



```
turtle(dragon(6))
```



Each successive curve consists of two copies of the preceding curve with a left turn between them; but the second copy is reversed, so that the turtle traces it from what was the end to what was the beginning.

An entertaining way of generating the dragon curve by hand is as follows: take a narrow strip of paper and lay it horizontally in front of you. Take the left-hand end of the strip and lay it over the right-hand end, thus folding the strip in half. Repeat this, taking the crease that is now at the left-hand end and placing it over the right-hand end, folding the strip in half again. Do this several times more, then open out the strip and make each crease into a right-angle. The edge of the strip will then form a dragon curve.

A different, but related, curve can be formed if, instead of performing all the folds from left to right, you alternate between folding the left end over the right and folding the right end over the left. Can you draw this curve with GeomLab?