Excursions in Computing Science: Week 12 Memory and Programming Language: Recursion and Instantiation

T. H. Merrett[*]
McGill University, Montreal, Canada

May 8, 2007

A. Recursion

1. Define "ancestor" in terms of "parent".

2. Precedence without parentheses

We'd like to parse `ab + cd` in the conventional way: `(ab) + (cd)`.

We can say `ab + cd` is an *expression* made up of *terms*.
"Made up" means combined by + (**or**).

Likewise, terms are made up of *identifiers* (letters), i.e., combined by adjacency, or sometimes • (**and**).

Grammar

```
<expression>::= <term> | <term> + <expression>
<term>::= <letter> | <letter><term>
```

These *recursive* definitions allow us to have indefinite numbers of terms with indefinite numbers of letters.

E.g., `a + bc + def + gh`

Let's see if we can use these recursive guidelines to write a program to recognize such an expression and put parentheses around the terms.

```
parsexp('ab+cd')

ans =

((ab)+(cd))
```

1. `parsexp` puts out a '(', calls `expression`, and puts out a ')'.

[*]Copyleft ©T. H. Merrett, 2006 Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation in a prominent place. Copyright for components of this work owned by others than T. H. Merrett must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to republish from: T. H. Merrett, School of Computer Science, McGill University, fax 514 398 3883. The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

2. `expression` puts out a ‘(’, calls `term`, puts out a ‘)’, calls `plussign` and, if `plussign` found a plus in the input, calls *itself* recursively to get the next expression.

3. `plussign` puts out ‘+’ and reports success if ‘+’ is indeed the next character of the input; otherwise it reports failure.

4. `term` calls `letter` and, if `letter` found a letter in the input, calls *itself* recursively to get the next term.

5. `letter` puts out the letter and reports success if the next character of the input is indeed a letter; otherwise it reports failure.

Here is `parsexp`. Note that MATLAB obliges us to keep explicit account of the input (`code`), our current input position (`inpos`), and the output (`parsed`) and our current output position (`outpos`).

```
% function parsed = parsexp(code)
% THM 060829
% uses expression.m
function parsed = parsexp(code)
parsed(1) = '(';
[parsed,inpos,outpos] = expression(code,parsed,1,2);
parsed(outpos) = ')';
```

Here is `expression`. I’ll leave `plussign, term` and `letter` as exercises.

```
% function [parsed,inpos,outpos] = expression(code,parsed,inpos,outpos)
% THM 060829
% used by parsexp.m; uses term.m, plussign.m; term.m uses letter.m
function [parsed,inpos,outpos] = expression(code,parsed,inpos,outpos)
parsed(outpos) = '('; outpos = outpos + 1;
[parsed,inpos,outpos] = term(code,parsed,inpos,outpos);
parsed(outpos) = ')'; outpos = outpos + 1;
[succ,parsed,inpos,outpos] = plussign(code,parsed,inpos,outpos);
if succ
 [parsed,inpos,outpos] = expression(code,parsed,inpos,outpos);
end
```

We can see that recursive thinking makes the process very much easier than without recursion.

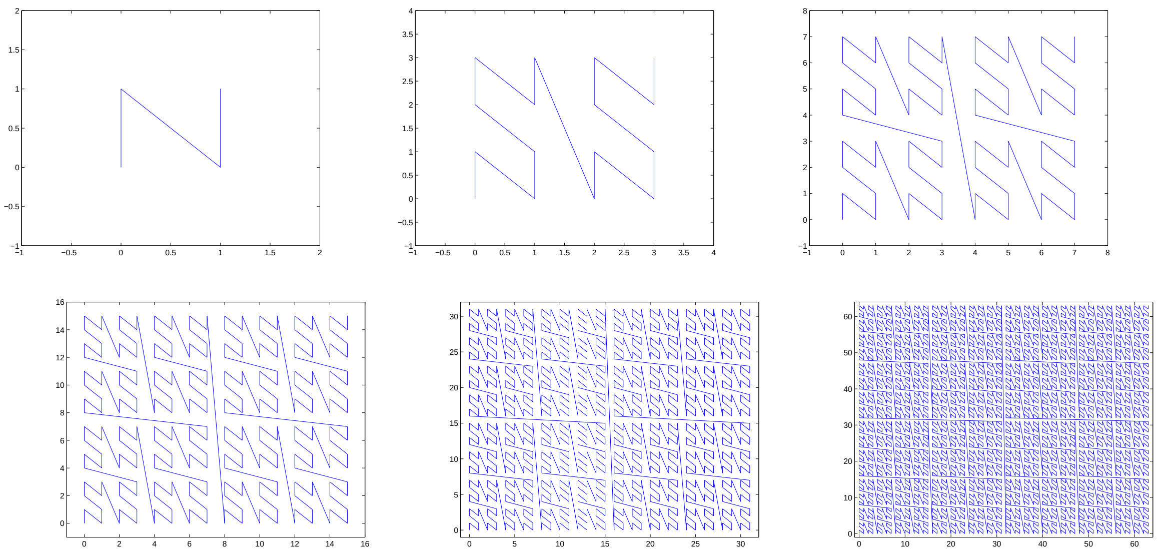
3. Fractals

A significant benefit of recursion is in computing an *exponential number* of things.

`Expression` and `term` only called themselves once, but if a function calls itself twice and does this recursively to depth n then we can get 2^n invocations of the function.

We don’t often *want* an exponential number of invocations because it is very expensive (see the Excursion this week about the chessboard), but one attractive application is drawing self-similar figures (*fractals*).

Here is the “Peano curve”, named after the French mathematician who published it in 1880 [Pea90]. I have run the program to depths 1–6.



1. `peano(n)` sets up the arrays that MATLAB needs to represent the final drawing, calls `peanoStep(n)`, and then uses MATLAB's `quiver()` and `axis()` functions to produce the drawing.

2. `peanoStep(n)` calls `peanoStep(n−1)` four times and also draws three lines, one in between each invocation: first vertically up then diagonally down and rightwards then vertically up again; the length of the lines depends on the current level, n.

Here is `peanoStep()`. Remember again that MATLAB obliges us to do all the housekeeping explicitly, including keeping track of the "screen" represented by the arrays X, Y, U and V.

```
% function [X,Y,U,V,j,k] = peanoStep(n,X,Y,U,V,j,k)
% THM 060829
% called from peano(n); uses draw.m
function [X,Y,U,V,j,k] = peanoStep(n,X,Y,U,V,j,k)
if n>0
  [X,Y,U,V,j,k] = peanoStep(n-1,X,Y,U,V,j,k);
  x =1-2^(n-1); y = 1; draw
  [X,Y,U,V,j,k] = peanoStep(n-1,X,Y,U,V,j,k);
  x = 1; y =1-2^n; draw
  [X,Y,U,V,j,k] = peanoStep(n-1,X,Y,U,V,j,k);
  x =1-2^(n-1); y = 1; draw
  [X,Y,U,V,j,k] = peanoStep(n-1,X,Y,U,V,j,k);
end
```

(A nice thing about recursive programming is that as you improve your code it gets *smaller*. My original version of `peanoStep()` had a stopping condition for $n = 1$. Why is this neither needed nor very good?)

(When the Peano curve is drawn in any number of dimensions (including 2-D as a special case), it is called "Z-order". It can actually be drawn, to fixed depth, without recursion by interleaving (or "shuffling") the bits that represent the coordinates of the grid points being connected by the Z-order.)

4. Mathematical induction.

A good way to think about recursion is as a form of proof by mathematical induction. This requires an *induction step*, which is the recursive call, and a *starting step*, which becomes the *stopping condition* in a recursive program.

Typically, mathematical induction is used to prove something is true for all integers, n, by showing

1. the "something" is true for $n = 1$;
2. if the "something" is true for $n - 1$, then it is true for n.

This is related to linear (non-exponential) recursion.

Another linear recursion is used to find the greatest common divisor of two integers (and the corresponding mathematical induction is proof that the recursion is correct):

```
function g = gcd(x,y)
  if y==0, g=x;                  %stopping condition
  else g = gcd(y,rem(x,y));      %recursion/induction step
```

Try this on $x = 38, y = 14$. Try it with paper and pencil: the sequence of calls is

step	0	1	2	3	4
x	38	14	10	4	2
y	14	10	4	2	0

We saw in week 10 that x and y have the same greatest common divisor (gcd) as y and x mod y. This is the induction step in a mathematical proof that the `gcd` function is correct, and it is the recusive step in the function. (The MATLAB function `rem(x,y)` and x mod y do the same thing. They give the remainder when integer x is divided by integer y.)

Furthermore, the recursion reduces the sizes of the parameters so they get smaller and smaller. Persuade yourself that y must eventually be 0, so that the x that made it so is an exact divisor of both the preceding x and y ... and so of the original x and y.

This was an inductive argument showing that the `gcd` program works. It is called Euclid's algorithm.

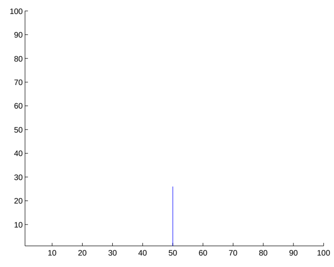
5. MATLAB syntax is not particularly elegant for recursion, and we can do better in other languages.

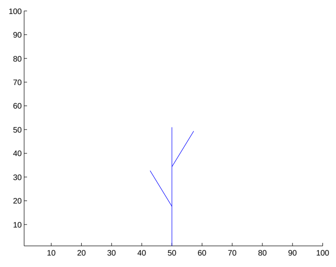
```
function gcd(x,y) is
  if y==0 then x
  else gcd(y,x mod y)
```

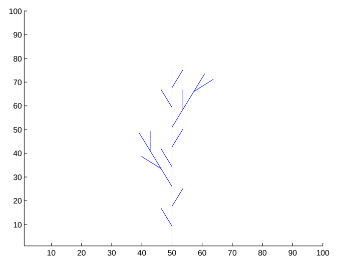
A programming language which supports only *expressions*, without needing *assignment statements*, is an example of *functional programming*. LISP was the first thorough example of this.

6. L-systems

Botany provides many illustrations of recursion: a branch produces sub-branches which produce sub-sub-branches, and so on.







Such botanical structures are also *self-similar*. How many times is the second "tree" drawn above found in the third? How does each of these correspond to a straight line in the second? Can you see what will happen next?

The basic pattern can be captured symbolically.

`B[+B]B[−B]B`

where

`B`: draw a line of given length
`+`: turn left by a given angle
`−`: turn right by a given angle
`[`: store current position and heading
`]`: retrieve stored position and heading

(`[` and `]` are *stack* operations, so several (position,heading) states may be stored, to be retrieved in the inverse order.)

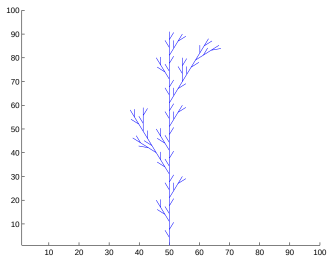
Now for the recursion. To make a self-similar version to the next degree of resolution, turn the symbolic string into a *rewriting rule*

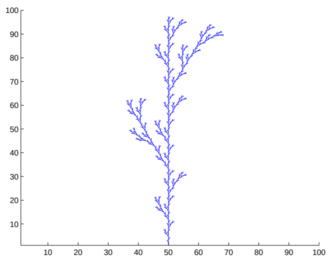
`B` $\longrightarrow$ `B[+B]B[−B]B`

This has the effect of replacing each `B` in the string by the string itself.

Applied an indefinite number of times, the rewriting rule "grows" a structure of arbitrary complexity.

Here are the next two steps following the three shown above.





MATLAB can implement this, although in a limited way. Using `quiver` to draw the final picture, we can build up the picture recursively in a way which follows the symbolic string.

1. `branchOL1(max,n)` initializes the size of the picture, given by `max`, initializes arrays needed by the stack, and then calls `branchOL1Step(n,..)`.

2. `branchOL1Step(n,..)` calls `branchOL1Step(n−1,..)` five times, corresponding to the five occurences of `B` in the rewriting rule, or, if $n = 0$, directly calls `forward()` to draw the straight line represented by the lowest level of `B`. `branchOL1Step()` uses `pushBOLstate()` and `popBOLstate()` to do the stack operations where indicated by [and], respectively.

Here is `branchOL1Step()`. (It must explicitly pass as parameters and return as results all the global variables needed for the current position and heading, and for the stacked positions and headings.)

```
% function [x,y,h,sX,sY,sH,sNext] =
   branchOL1Step(n,step,delta,x,y,h,sX,sY,sH,sNext)
% THM 061023 in file: branchOL1Step.m
% called by branchOL1.m; uses forward.m popBOLstate.m, pushBOLstate.m
function [x,y,h,sX,sY,sH,sNext] =
   branchOL1Step(n,step,delta,x,y,h,sX,sY,sH,sNext)
if n==0, [U,V,x,y] = forward(step,U,V,x,y,h); else
```

```
  [x,y,h,sX,sY,sH,sNext] = branchOL1Step(n-1,step,delta,x,y,h,sX,sY,sH,sNext);%B
  [sX,sY,sH,sNext] = pushBOLstate(x,y,h,sX,sY,sH,sNext); %[ PUSH
  h = h + delta; %+ left(delta)
  [x,y,h,sX,sY,sH,sNext] = branchOL1Step(n-1,step,delta,x,y,h,sX,sY,sH,sNext);%B
  [x,y,h,sX,sY,sH,sNext] = popBOLstate(sX,sY,sH,sNext); %] POP
  [x,y,h,sX,sY,sH,sNext] = branchOL1Step(n-1,step,delta,x,y,h,sX,sY,sH,sNext);%B
  [sX,sY,sH,sNext] = pushBOLstate(x,y,h,sX,sY,sH,sNext); %[ PUSH
  h = h - delta; %- left(-delta)
  [x,y,h,sX,sY,sH,sNext] = branchOL1Step(n-1,step,delta,x,y,h,sX,sY,sH,sNext);%B
  [x,y,h,sX,sY,sH,sNext] = popBOLstate(sX,sY,sH,sNext); %] POP
  [x,y,h,sX,sY,sH,sNext] = branchOL1Step(n-1,step,delta,x,y,h,sX,sY,sH,sNext);%B
end
```

B. Instantiation

Comparing recursive with nonrecursive code, we see that the parameters of the recursive function provide a kind of *workspace.*

Recursive

```
function gcd(x,y) is
  if y==0 then x
  else gcd(y,x mod y)
```

Iterative

```
function gcd(x,y)
  while(y>0)
  { y' = x mod y;
    x = y;
    y = y';
  }
  return x;
```

(This is not MATLAB code.)

Notice that the iterative (nonfunctional) code, which *assigns* values to variables `y'`, `x` and `y`, must have an additional "workspace", `y'`. The recursive code just uses the parameters to serve as this workspace.

But there is a kind of workspace which functional parameters cannot capture.

This is any variable whose value must be kept *between invocations* of a function.

This is the case for most of the variables in the functions in this week's notes, and we see what a pain MATLAB gives us over them. But all of the examples so far this week need to make the values of these variables available *outside* the functions as well as inside them.

The flipflop program of Week 11 is different. The "state" of the flipflop needs to be known only by the flipflop. The functions that call the flipflop, such as `flipflopRead()` and `flipflopWrite()`, do not need to know the state, say `y`, as we said in Note 1 of Week 11. MATLAB obliged us to write

```
function y = flipflopWrite(data,y)
```

when it should only be necessary to write

```
flipflopWrite(data)
```

We can do this better (but not in MATLAB) by maintaining `y` as a *state* which is held over between invocations—but not available to any code outside of the function for which it is intended.

If we had such a capability, here is how we might wrap up the flipflop function we already wrote in Week 11.

```
statefunction flipflopstateWrite(data)
  state y;
  y = flipflopWrite(data,y)
```

```
end flipflopstateWrite
```

(This is not MATLAB code.)

This is incomplete. `y` is not *initialized* for one thing. We could just say `state y = 1;` to do it.

More importantly, the state cannot be *shared* with another function, say `flipflopstateRead()`.

So we need to *encapsulate* the state and the functions we need to work with it.

```
flipflop
  state y = 1;
  function flipflopstateWrite(data);
  % define the code here (as above)
  function data = flipflopstateRead();
  % define the code here (similarly)
end flipflop
```

(This is not MATLAB code.)

So we could now invoke the flipflop functions in this way:

```
 flipflopstateWrite(1);
 ..  = flipflopstateRead();  % gets 1
 ..  = flipflopstateRead();  % gets 1
 :                           :
 flipflopstateWrite(0);
 ..  = flipflopstateRead();  % gets 0
 :                           :
```

8. What if we wanted *two* flipflops?

We'd have to make a *copy* of the state.

(We don't have to copy the *functions*: they are just code. This is for the same reason that we didn't have to copy the `nand()` function when we built the flipflop in the first place. We just used it, repeatedly.)

We can take our above definition of `flipflop` as a template for a *class* of flipflops. If we provide some new syntax, the programmer can *instantiate* as many flipflops as hey likes.

```
 f1 = new flipflop;
 f1.flipflopstateWrite(1);
 ..  = f1.flipflopstateRead();  % gets 1
 :                              :
 f2.flipflopstateWrite(0);
 ..  = f1.flipflopstateRead();  % gets 1
 ..  = f2.flipflopstateRead();  % gets 0
 :                              :
```

Classes and their instantiation are the heart of (the very badly named) "object-oriented" programming. In fact, although you will hear many opinions about "O-O" programming, only instantiation fundamentally matters.

This kind of programming violates the goal of functional programming, which is to dispense altogether with state, or any hidden "side effects" caused by assignment and update operations.

But it is an improvement over unrestricted states and side effects. It *contains* each state within the module that also contains all the allowed ways of operating on that state. This is called *encapsulation*.

MATLAB can't do any of the coding we've just seen. Can we cobble together a state-preserving, instantiable flipflop in MATLAB?

Instead of `flipflop`, for which we have already written a whole lot of code which we'd have to modify, let's try the most basic possible class, `counter`.

```
counter
  state ctr = 0;

  function reset
       ctr = 0;
  end reset

  function count
        ctr = ctr + 1;
        return ctr;
  end count
end counter

sheep = new counter;
goats = new counter;
.. = sheep.count;
.. = sheep.count;
.. = goats.count;
sheep.reset;
:
```

We can do this in MATLAB but it will be much longer: we must manage the names and the states ourselves. We can do this using arrays to remember the various instances of the state.

9. Summary

(These notes show the trees. Try to see the forest!)

- Recursion
  - Grammar: processing and evaluating expressions.
  - Fractals: self-similar curves with exponential complexity.
  - Mathematical induction.
  - L-systems: self-similar branching structures.
- Instantiation
  - Workspace vs. state.
  - Functional vs object-oriented programming:
    - functional programming has no side-effects (e.g., assignments), and hence no *state*;
    - object-oriented programming has state, and *instantiation* to copy the state, but it *encapsulates* the state to make it safer;
    - conventional programming (usually called "imperative") allows assignments and state anywhere, thus forcing the programmer to remember all the assigned variables and their values at any given point in the program.

10. **Excursions for Friday and beyond.**

You've seen lots of ideas. Now *do* something with them!

1. Combine parsexp with the shuntingYard algorithm and the booleanRevPolEval function of Week 11 to parse and evaluate expressions such as ab+cd.

2. How must the grammar implemented by parsexp be changed to accommodate parentheses in the *input*, such as in a(b+c)d?

3. How many function calls are generated by a function which calls itself thrice and recurses to depth n (the initial call being at level 1, the next two at level 2, and so on to level n).

4. The inventor of chess is legendarily said to have been invited by his grateful head of state to name any reward he wished. He asked for one grain of wheat for the first square of the chessboard, two for the second, double that for the third, double it again for the fourth square, and so on. How many bushels of wheat did the monarch have to produce?

5. Use MATLAB to calculate the interest growth formula, $v = v_0(1+i)^p$, for the value, v, of a quantity which has grown at interest $i \times 100\%$ per period over p periods. For small interest rates of 1%, 2%, 5%, .., compare this with e^{ip}. Given that $e^{0.72} \simeq 2$, what is a quick way to find out how many years it takes for your money to double if invested at small interest rates?

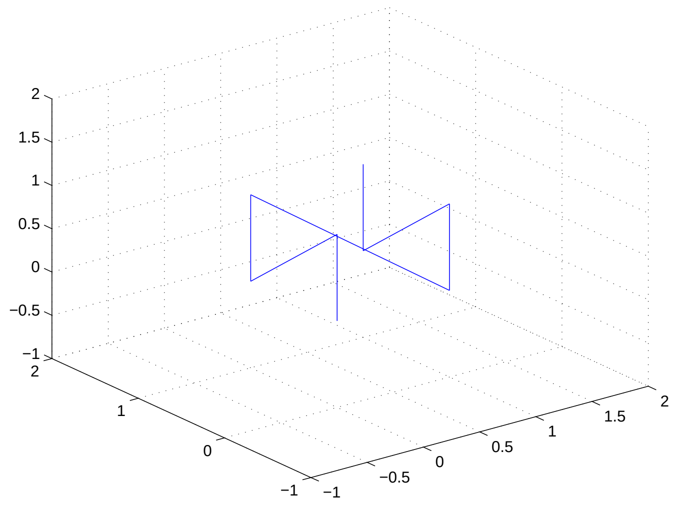
6. The following table gives the gross domestic product (GDP) per capita in 1820 for various regions of the world, and the annual percentage growth rate for each of these regions. (I have interpolated this data from [Sac05, Chap. 2], who cites his sources.)

Region	Western Europe	Eastern Europe	Former USSR	US, Canada Oceania	Latin America	Japan	Asia (not Japan)	Africa
1820 GDP/cap.	1500	800	800	1200	800	800	800	800
% growth/ann.	1.5	1.2	1.0	1.7	1.2	1.9	0.9	0.7

   Use MATLAB to calculate the GDP per capita in 2000, 180 years later, and discuss the poverty gaps.

7. Rewrite peanoStep with a stopping condition for $n == 1$. Why is this not as good?

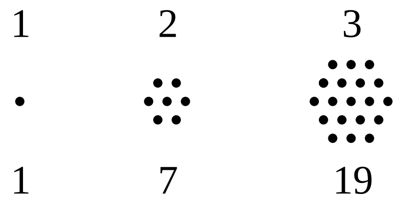
8. Write a three-dimensional Peano curve drawing program. Here is the basic unit.



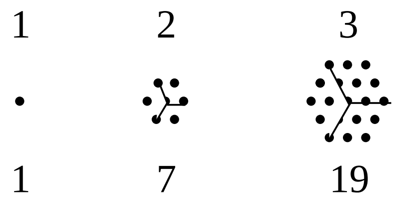
9. Write a two-dimensional Z-order program to draw the Peano curve not using recursion but by interleaving bits of the coordinates.

10. Look up the "Hilbert curve" and write a MATLAB program to draw it to any depth of self-similarity.

11. *Hexagonal numbers*, 1, 7, 19, 37, 61, 91, 127, .., are the numbers of dots that can be used to show filled hexagons of sides 1, 2, 3, ..



a) Show by mathematical induction that the sums of hexagonal numbers up to any point, starting at 1, are always perfect cubes (1, 8, 27, 64, ..).
b) Show the same, but this time by visualizing each hexagon as the front three faces of a cube, whose interior can be seen to be filled by all the preceding hexagons.



c) Penrose [Pen94, pp.66–77] argues that proofs such as the above visualization cannot be captured by computational rules such as induction—a visualization can always be found which transcends any currently complete computational system—and hence that mind cannot be programmed. (See the Excursion in Week 11 on non-stopping computations.) Don't take my word for it: read the whole book!
Penrose's argument could explain AI's apparent lack of significant progress, over half a century, towards its main goal, the construction of an intelligent program. AI has, however, provided significant benefits to computer science, in the LISP and Prolog programming languages, the idea of expert systems, techniques used in data mining, and many other paradigms. Penrose does believe that an intelligent machine can be constructed—it just cannot be a program based on current computers.

12. Our program for gcd is not totally safe. It needs to be protected against $x < y$ and against negative inputs. Using only if statements and further calls to gcd itself, add these protections to the code.

13. Express factorial as a recursive function. What are the starting and inductive steps needed to prove that your approach is correct?

14. Look up the legend of the "Towers of Hanoi" and write a recursive program to end the Universe.

15. Look up John McCarthy (1927–) and the functional programming language LISP. How can the syntax of a language be identical to the syntax of the data it processes? Write a LISP function to reverse the elements of a list.

16. Rewrite the branch-drawing program of Note 6 using `gplot()` in MATLAB.

17. Look up Prusinkiewicz and Lindenmayer's *The Algorithmic Beauty of Plants* [PL90] and write MATLAB programs for some of the other branching structures on p.25.

18. Write a translator to convert the symbolic strings, that give rewriting rules, into programs to draw the recursive pictures. (Advanced skills and a better language than MATLAB are required.)

19. Write a set of MATLAB functions to mimic the object-oriented Counter class without using arrays as storage. Explain why not using the arrays is a little misleading in a general discussion of the advantages of automatic instantiation.

20. Any part of the lecture that needs working through.

References

[Pea90] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.*, 36:157–60, 1890.

[Pen94] Roger Penrose. *Shadows of the Mind.* Vintage (Random House), London, 1994.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, New York, 1990. Electronic version 2004: algorithmicbotany.org/papers/#abop.

[Sac05] Jeffrey D Sachs. *The End of Poverty: Economic Possibilities for Our Time.* Penguin Group (USA) , Inc, New York, 2005.