How to hack in TEX

Maarten Pennings

june 1990

Abstract

This paper shows my first hack-experiments in TEX. Understanding LATEX and having some programming experience should suffice to understand the basic concepts. Then you can hack your way in TEX.

1 Introduction

I suppose you have some basic knowledge about LATEX. For all experiments in this paper you only need TEX, the Leslie Lamport macros only slows the edit-compile-view cyclus down so I used TEX — not even plain TEX. Type `TEX <myfile>` to use TEX, instead of `TEX &PLAIN <myfile>` for plain TEX, or `TEX &LPAIN <myfile>` for LATEX. And don't forget to finish your TEX-file with a `\bye`!

1.1 Macros

All LATEX commands can be expressed by TEX commands. Sometimes you need some unreadable lengthy macro — like the ones we are going to develop — but another time the coding is straightforward. As an example of the latter the `\newcommand{`*name*`}[`*num*`]{`*definition*`}` serves us well as we will use it frequently. It defines a new command with the name *name*.

If you talked in class and you got caught, you must write 100 times "I must not talk in class.". You could solve this by the following macros:

```
\newcommand{\five}[1]{#1\par #1\par #1\par #1\par #1\par}
\newcommand{\twen}[1]{\five{#1} \five{#1} \five{#1} \five{#1}}
\newcommand{\hundred}[1]{\twen{#1} \twen{#1} \twen{#1} \twen{#1} \twen{#1}}
\hundred{I must not talk in class.}
```

And this is a rather good way to do it, because if you get caught running in the halls later on your punishment is even less work: `\hundred{I must not run in the halls.}`.

But we were going to do things the TEX-way. The TEX equivalent of `\newcommand` is `\def`*name parameters*`{`*definition*`}`. The above macros are equivalent to

```
\def\five#1{#1\par #1\par #1\par #1\par #1\par}
\def\twen#1{\five{#1} \five{#1} \five{#1} \five{#1}}
\def\hundred#1{\twen{#1} \twen{#1} \twen{#1} \twen{#1} \twen{#1}}
\hundred{I must not talk in class.}
```

When you pass parameters from one macro to another in this way, you punish TEX much harder than you need to, because it takes a lot of time to copy parameters and read them again and again. It is much faster to define a control sequence:

```
\def\five{\one \one \one \one \one}
\def\twen{\five \five \five \five}
\def\hundred#1{\def\one{#1\par} \twen \twen \twen \twen \twen}
\hundred{I must not talk in class.}
```

You think you fully comprehend macro expansion? Then try to solve the following puzzle, stolen from the TEX-book: What is the expansion of `\puzzle` given (for answer see last page):

```
\def\a{\b}
\def\b{A\def\a{B\def\a{C\def\a{\b}}}}
\def\puzzle{\a\a\a\a\a}
```

1.2 Counters

I hope you agree with me that the solution to the 100 line of punishment is artificial. And, if you have a hateful teacher, he might come up with a large prime number, making your task as macro-designer even harder (Tip: if n is prime, solve the $n-1$ (no prime!) puzzle and print a single extra line).

It would be far more elegant to have another parameter — a number — that specifies how many lines you have to write. We need a counter for that. LATEX says `\newcounter{`*counter*`}` to introduce an integer, but we will use the TEX-way again: `\newcount`*counter*.

To initialise such an integer LATEX says `\setcounter{`*counter*`}{`*value*`}` whereas TEX says *counter*`=`*value*. To "increase" an integer we have `\addtocounter{`*counter*`}{`*value*`}` in LATEX and `\advance`*counter* `by` *value* in TEX.

There is one more thing we have to know before we can write a recursive program: selection (recursion needs to terminate). The syntax for a comparison between two integers is: `\ifnum`*counter rel value commands*`\fi`. Of course, this executes *commands* if and only if the guard holds. The guard consists of a counter (`\n`), a relation which is one of `<`, `=` and `>` and a number (or counter).

As you will see in the example below, I have chosen a small prime: 13. Why? Well, if you take into account how much has to be pushed onto the stack at each call, you should not be surprised to hear that you get a `TeX capacity exceeded, sorry` quite soon. For large punishment you should use repetitions, I will introduce them below.

The following recursive program prints your opinion on teachers 13 times:

```
\newcount\n                    % Var n:Integer;
\def\Punish#1#2%               % Procedure Punish(p1,p2);
   {                           % Begin
   \n=#1                       %   n:=p1;
   \ifnum\n>0                  %   If n>0 Then Begin
     #2\par                    %     WriteLn(p2);
     \advance\n by -1          %     n:=n-1;
     \Punish{\n}{#2}           %     Punish(n,p2)
   \fi                         %   End
   }                           % End
\Punish{13}{I hate prime numbers, and teachers who use them.}
```

The type of the parameters is not specified. The user may pass anything, for example `\Punish{xx}{test}` will not generate an error on calling the macro. The command `\n=#1` however will generate an error since `\n=xx` is illegal.

You should not type `\ifnum\n>0#2` etc but `\ifnum\n>0 #2` etc, because if you omit the space after the constant, TEX will expand the next token in order to see whether it starts with some digits. Suppose `#2='11 june 1990'` then the guard reads `\ifnum\n>011 june 1990` so you get 11 lines less than you probably expected (since `011` is a legal constant).

A final remark about the program: the `\def\Punish#1#2` should be followed by a `{`. For formatting purposes, we placed this curly brace on the next line, therefore we had to suppress the current end-of-line with a `%` immediately following the `..#1#2`. There are, however, cases in which we should append a `%` to a line where leaving one out does not cause an error. These cases do cause unwanted space in the output. And these spaces are quite hard to trace. As a rule of thumb: a line that ends with a control sequence (such as `\fi`), a dimension (`pt`) or with a constant (such as `\advance\n by -1`) "eats up" the trailing spaces and therefore does not need a `%`. The `{` and `\Punish{\n}{#2}` typically do introduce spaces, but in our macro they are not harmful (see Hanoi problem for a real-life version).

1.3 More on counters

I already told you about basic counter-handling stuff such as `\newcount`, `=` and `\advance by`. Counters can also be multiplied and divided. The following fragment computes `m mod 10`

```
\n=\m                   % n = m
\divide\n by 10         % n = m div 10
\multiply\n by -10      % n = -10(m div 10)
\advance\n by \m        % n = m - 10(m div 10)
                        %   = 10(m div 10) + m mod 10 - 10(m div 10)
                        %   = m mod 10
```

Counters are 32 bits integers; they have range [−2.147.483.647 , 2.147.483.647]. When defined, they are automatically initialised to zero. To convert a counter `\n` to printable characters use `\number\n`.

For fun, try `\romannumeral\n`. And, fellow-hackers, it is much harder to convert a counter to uppercase roman-characters — even when you know there exists an `\uppercase`. The following should do the trick, the TEX-book claims, don't ask *me* why:

```
\uppercase\expandafter{\romannumeral\n}
```

Note that `\uppercase{a\lowercase{bC}}` has the peculiar result of `A\lowercase{BC}` and thus `Abc`. `\expandafter\a\b` first expands `\b` and then `\a` so let me know if this makes sense to you. (TEX-book hint: the `\expandafter` expands the token after the `{`, not the token after the group.)

In the previous section I warned you not to forget a space after a constant. And I gave an example, but it was a lousy one. The next one is more advanced. What do you think this fragment generates:

```
\newcount\n        % auto init to 0
\def\a{\number\n:}
\n=3\a\a
```

It is *not* `3:3:`, for if it were, there would be no need to warn you, would there? No, the result is `:30:`. Surprised? Well, the constant `3` might be extended by the following macro (since the `\a` immediately follows the `3` — there is no space), so TEX expands the first `\a` to check for this. And indeed, it expands to `0:` (since `\n` is auto-initialised to zero), so the `0` is appended to the `3` — the assignment sets `\n` to `30` — and a colon is printed. Then `30:` is printed by the next `\a`.

1.4 Repetition

In the previous subsection we already saw the `\if \fi` construct. In this subsection we will see some more constructs. An `\if \fi` may be extended with an `\else` clause as in

```
\ifnum\n>\m \biggest=\n
\else       \biggest=\m
\fi
```

TEX also allows many-way branches (also known as `case`), which compares best to the BASIC `ON i GOTO 1000,2000,3000,...` construct. The cases are counted from zero so, with `\dayofweek` as defined below, `\dayofweek3` expands to `thursday`. By the way, note the lack of braces here, which is allowed with single character arguments as you may remember from, for example, the LATEX `\"e` command.

```
\def\dayofweek#1{%
  \ifcase#1 monday%
  \or       tuesday%
  \or       wednesday%
  \or       thursday%
  \or       friday%
  \or       saturday%
  \or       sunday%
  \else     illegal%
  \fi
  }
```

And finally — what is a programming language without a repetition (if you agree with this you are too imperative minded, we have recursion!) — repetition. It is illustrated by the following algorithm that computes factorials:

```
\newcount\f
\newcount\n
\def\fac#1{%
  \f=1 \n=#1
  \loop\ifnum\n>0
     \multiply\f by \n
     \advance\n by -1
  \repeat
  \number\f
  }
\fac{10}
```

Note that the `\if` has no corresponding `\fi`; the `\repeat` delimits its scope. Between the `\loop` and the `\if` you may put some commands that should be executed regardless of the value of the guard. So this `\loop \repeat` is, in fact, a mixture of the PASCAL `while` and `repeat` loops.

2 Fibonacci

To make you enjoy reading this paper, I will now introduce a more elaborated example, using most of the previous topics. The algorithm computes the first n ($n \geq 2$) fibonacci numbers. Their formal definition is:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_{n+2} &= f_{n+1} + f_n \end{aligned}$$

As you may know, the above function does not immediately lead to an efficient algorithm to compute them. The number of function calls is proportional to the functionvalue itself,

and, fibonacci numbers grow exponential. Although logarithmic algorithms exist, we stick to the simple linear one:

```
\newcount\n                 % Var n:Integer;
\newcount\a                 % Var a:Integer;
\newcount\b                 % Var b:Integer;
\def\fib#1%                 % Procedure fib(param1:Integer);
{%                          % Begin {N=param1>1}
  \n=#1 \advance\n by -2    %   n:=param1; n:=n-2;
  \a=0 \b=1                 %   a:=0; b:=1; {a=Fib(N-n-2) and B=Fib(N-n-1)}
  \number\a                 %   Write(a);
  \loop\ifnum\n>0           %   While n>0 Do Begin
    ,                       %     Write(', ');
    \number\b               %     Write(b); {a=A and b=B}
    \advance\b by \a        %     b:=b+a;   {a=A and b=A+B}
    \multiply\a by -1       %     a:=-a;    {a=-A and b=A+B}
    \advance\a by \b        %     a:=a+b;   {a=-A+A+B and b=A+B}
    \advance\n by -1        %     n:=n-1    {a=B and b=A+B}
  \repeat                   %   End;
  ~and~\number\b            %   Write(' and ',b)
}                           % End;

The first 20 fibonacci numbers are \fib{20}.
```

The first 20 fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584 and 4181. Since the 47th fibonacci number is larger than 2^{31} the above program generates an arithmetic overflow error on `fib{`n`}`, for $n \geq 47$.

3 Scopes

In this section I will discuss some scope-related topics. You don't need to read this section to understand the Hanoi program discussed next.

In the fibonacci example, it would have been nicer to have the variables declared local to the procedure. And, in fact this is possible: the three `\newcount` commands could have been placed within the `\fib` scope (between `{` and `}`). But even when making these declarations local, they remain global according to TEX conventions (besides, the braces in a `\def` are macro-delimiting braces; not the usual scope delimiters).

In the next example, the integer `\n` is declared locally, but once declared, it remains "active", i.e. its scope is global:

```
{               % start a new scope
  \newcount\n   % note: auto-initialisation to 0
}               % close scope
```

```
\number\n       % counter n still exists!
```

A comment on the previous program for Real Hackers. The open-brace does PUSH the current state space, while `}` POPs it. Thus

```
\newcount\n     % note: auto-initialisation to 0
{               % start a new scope, \n=0 is pushed
  \n=3          % set \n to 3
  \number\n     % generate a \n (a '3')
}               % close scope, \n=0 is popped
\number\n       % generate a '0'
```

You should remember that the braces in the LaTeX `\newcommand` or TeX `\def` commands do not belong to the command defined and therefore do not delimit its scope. The above example is *not* equivalent to

```
\def\example{\n=3 \number\n} % braces delimit macro-def, not the scope.
\newcount\n                  % note: auto-initialisation to 0
\example                     % macro as in the above example
\number\n                    % generates a '3'!
```

but to

```
\def\example{{\n=3 \number\n}} % scope braces included.
\newcount\n                    % note: auto-initialisation to 0
\example                       % macro as in the above example
\number\n                      % generates a '0'
```

It gets really hacking now. Suppose you are designing a print-pagenumber macro, you might get into problems with the above scope conventions. For a command such as `\n=1` to set the current pagenumber to 1 might not have this effect globally: if it is executed within scope delimiting braces! To get around this use `\global`

```
\newcount\n     % note: auto-initialisation to 0
{               % start a new scope
  \global\n=3   % set \n to 3 (locally and globally)
  \number\n     % generate a \n (a '3')
}               % close scope
\number\n       % generate a '3'
```

The trick used here is that `{` does not really push the state space, as this would cost far too much memory (do you know how large the TeX state space is?). It is a command such as `\n=3` that pushes the value of `\n`, when it appears between braces. And upon closing the scope (`}`), all changes are popped. And now the trick is straightforward; `\global\n=3` does

not push anything. If you would like to monitor this popping, set `\tracingrestores=1` and examine the logfile.

You think you fully understand the scope rules? Then tell me, what values will be output by the following fragment of global and local assignments:

```
\newcount\n
\def\g{\global\n=}
\def\l{\n=}
\def\s{ \number\n}
{\l1\s\g2{\s\l3\s\g4\s\l5\s}\s\l6\s}\s
```

(for answer see last page)

4 Hanoi

In this section we will develop a macro that prints the start and end situation of the Towers of Hanoi problem and all legal one-step situations in between. It gets really messy, so we build it up gradually.

We start with a simple Pascal program that only prints the moves. From this we infer the TEX macros. Then we introduce a (tricky) data structure that represents the three piles and the `\move` operator that moves a disk from one pile to another. We then incorporate this into our program. First we print the piles horizontally, then vertically. Finally we design nice looking blocks to present the steps performed.

4.1 Pascal

The following program generates a list of actions to move a 4 disk pile from pin 1 to pin 2 (having pin 3 as spare), according to the rules of the Towers of Hanoi. I hope you remember the Pascal procedure has the following form:

```
Procedure Hanoi(n,a,b,c:Integer);
  Begin {Hanoi}
    If n>0 Then Begin
      Hanoi(n-1,a,c,b);
      Writeln('(from ',a,' to ',b,' )');
      Hanoi(n-1,c,b,a)
    End
  End;  {Hanoi}

Begin {Main}
  Hanoi(4,1,2,3)
End.  {Main}
```

"The Towers of Hanoi form a time consuming puzzle. To solve the 4 disk puzzle you need the following moves: (from 1 to 3) (from 1 to 2) (from 3 to 2) (from 1 to 3) (from 2 to 1) (from 2 to 3) (from 1 to 3) (from 1 to 2) (from 3 to 2) (from 3 to 1) (from 2 to 1) (from 3 to 2) (from 1 to 3) (from 1 to 2) (from 3 to 2)." is produced by the TEX translation of this PASCAL program:

```
\newcount\n

\def\Hanoi#1#2#3{\ifnum\n>0
  \advance\n by -1
  \Hanoi{#1}{#3}{#2}
  (from~#1~to~#2)%
  \Hanoi{#3}{#2}{#1}%
  \advance\n by +1
  \fi}

\def\TowersOfHanoi#1{\n=#1 \Hanoi{1}{2}{3}}

The Towers of Hanoi form a time consuming puzzle. To solve the 4 disk
puzzle you need the following moves:\TowersOfHanoi{4}.
```

Remember the insert-spaces-after-constant-remark? Remove the space between `\n=#1` and `\Hanoi{1}{2}{3}` and you won't get any output at all! And remember the append-space-suppressing-`%`-remark. Well, leave out the `%`s and you get unevenly distributed spaces between your moves. Now, you get a space *before* every move.

4.2 A pile

We do not want to print the moves (the state transformations) but the three piles (the states) themselves. For this we need a datastructure that represents a pile of disks of different sizes. As far as I know, lists, arrays or stacks are not available in TEX.

But hey, we are hackers, we can use 31 bits integers for that (the 32^{nd} bit is the sign bit, which we will not consider). If we divide an integer in groups of three bits, we have 31 div 3 = 10 groups. This models and `array[0..9] of 0..7`, which represents a 10 stock pile that can hold disks of size 0 to 7.

The indexrange is a little bit large for our purposes: if we have 7 disks, we do not need 10 stock piles. The next step, however, divides the 31 bits in 7 groups of 4 bits. This is even worse: we then have 16 sizes disk whereas a pile can hold 7 disks max! Besides, the Towers of Hanoi game with a pile of n disks takes $2^n - 1$ moves, which gets fairly large for large n, so the 3 bit groups will do fine.

As an example consider the pile value $((4 \times 8 + 3) \times 8 + 2) \times 8 + 1 = 2257$, written 100011010001 in the binary system. If we divide this in groups of three bits, starting from the right we get:

```
                    2257
100   011   010   001
  4     3     2     1
```

Hence, we declare the "datastructures" representing the piles with the simple `\newcount` commands

```
\newcount\TowerOne
\newcount\TowerTwo
\newcount\TowerThree
```

The only operator we have on piles is move the upper disk from one pile to another. As the upper disk is represented by the three least significant bits of the pile a `mod 8` (see section 1.3) gets the size of the upper disk whereas `div 8` removes it. We add a disk by shifting the pile 3 bits left and adding the width of the disk. For hackers this should be a piece of cake.

```
\newcount\d       % a scratch variable
\def\Move#1#2{%
  \d=#1 \divide\d by 8 \multiply\d by -8 \advance\d by #1%
  \divide#1 by 8
  \multiply#2 by 8 \advance#2 by \d}
```

We have to initialise the three piles with the values [empty], [empty] and $1, 2, \ldots, n$ respectively. Empty is simple, just assign zero (all disks on the pile have width zero). The third pile is filled by repeatingly using the same shift-3-bits-left-and-add-trick as above. The code for the initialisation is then straightforward. `\PrintTowers` is the macro that we are waiting for, it will print the three towers; we use it here to print the start configuration.

```
\def\TowersOfHanoi#1{%
  \TowerOne=0 \TowerTwo=0 \TowerThree=0 \n=#1%
  \loop\ifnum\n>0
    \multiply\TowerOne by 8
    \advance\TowerOne by \n
    \advance\n by -1
  \repeat
  \n=#1 \PrintTowers
  \Hanoi{\TowerOne}{\TowerTwo}{\TowerThree}}
```

Note that we use a non-standard way to pass parameters. Not the value of `\TowerOne` is passed to `Hanoi` as first parameter, but this name itself (is this *call by name?*). All stuff scraped together yields the following program

```
\newcount\TowerOne
\newcount\TowerTwo
```

```
\newcount\TowerThree

\newcount\d
\newcount\n

\def\PrintTowers{%
  (\number\TowerOne~--~\number\TowerTwo~--~\number\TowerThree)}

\def\TowersOfHanoi#1{%
  \TowerOne=0 \TowerTwo=0 \TowerThree=0 \n=#1%
  \loop\ifnum \n>0
    \multiply\TowerOne by 8
    \advance\TowerOne by \n
    \advance\n by -1
  \repeat
  \n=#1 \PrintTowers
  \Hanoi{\TowerOne}{\TowerTwo}{\TowerThree}}

\def\Move#1#2{%
  \d=#1 \divide\d by 8 \multiply\d by -8 \advance\d by #1%
  \divide#1 by 8
  \multiply#2 by 8 \advance#2 by \d}

\def\Hanoi#1#2#3{\ifnum\n>0
  \advance\n by -1
  \Hanoi{#1}{#3}{#2}
  \Move{#1}{#2}%
  \PrintTowers%
  \Hanoi{#3}{#2}{#1}%
  \advance\n by +1
  \fi}

And as a result we have \TowersOfHanoi{3}.

\bye
```

And as a result we have (209 – 0 – 0) (26 – 1 – 0) (3 – 1 – 2) (3 – 0 – 17) (0 – 3 – 17) (1 – 3 – 2) (1 – 26 – 0) (0 – 209 – 0). If we convert these cryptic numbers (first line) to the binary system and chop them in groups of three bits (second line) we get more sensible results (third line). Note that piles are seperated by a single vertical line and two consecutive states by a double one.

$$\begin{array}{rrr|c|c\|rrr|c\|rrr|c\|}
 & & 209 & 0 & 0 & & 26 & 1 & 0 & 3 & 1 & 2 & \\
011 & 010 & 001 & 000 & 000 & 011 & 010 & 001 & 000 & 011 & 001 & 010 & \ldots \\
3 & 2 & 1 & – & – & 3 & 2 & 1 & – & 3 & 1 & 2 &
\end{array}$$

4.3 Vertical printing

The output results of our previous program were not too impressive. Some cryptic numbers that had to be converted to the binary system and then chopped in groups of three bits. We have a computer for that. So hackers, there is work to do. Print the disks, not the piles — vertically if possible. And of course this is possible.

We will develop a macro that prints a single tower, let's call it `\Print#1` where the parameter is, for example, the 209 from above, and the result is a box that contains `1 2 3`

```
\def\Print#1{%
  \e=#1%
  \loop\ifnum\e>0
    \d=\e \divide\d by 8 \multiply\d by -8 \advance\d by \e
    \divide\e by 8
    \number\d
  \repeat}
```

How the hell do we get this vertically? Tabulars, matrixes? Well, no, this TeX, a typesetting system. You want to have the boxes stacked vertically, `\vbox` the `\hbox`es, that's all there is to it:

```
\newcount\e        % scratch variable
\def\Print#1{%
  \e=#1%
  \vbox{%
    \loop\ifnum\e>0
      \d=\e \divide\d by 8 \multiply\d by -8 \advance\d by \e
      \divide\e by 8
      \hbox{\number\d}%
    \repeat
  }}
```

If we now change the `\PrintTowers` routine to

```
\def\PrintTowers{%
  (\Print{\TowerOne}~--~\Print{\TowerTwo}~--~\Print{\TowerThree})}
```

We have as solution, which looks already much better:

1							1
2	2		1	1		2	2
(3 – –)	(3 – 1 –)	(3 – 1 – 2)	(3 – – 2)	(– 3 – 2)	(1 – 3 – 2)	(1 – 3 –)	(– 3 –)

4.4 The finishing touch

In this section we will smooth out the `\Print#1` macro. It will not print numbers but disks! And, in fact, that is not much harder. If we define the new dimensions

```
\newdimen\Disk \Disk=0.13 cm
\newdimen\dDisk \dDisk=2\Disk
```

that define the thickness and width of a unit disk (a disk of size 1) then the command

```
\hbox{\number\d}%
```

in `\Print#1` should be replaced by

```
\hbox{\vrule width\d\dDisk height1\Disk depth0\Disk}
```

and we get disks instead of numbers! The `\vrule` command is like the LaTeX `\rule` command except that it also allows the rule to extend below the baseline (the `depth`).

We are now doing real I/O so things get really messy. I want, for example some (vertical) whitespace between individual disks. We get that right now, but it is the normal `\interlineskip`, that TeX puts between individual lines. If you decrease `\Disk` this space is too large, if you increase `\Disk` it is too small.

So, suppress the normal whitespace (`\nointerlineskip`) and add your own. For the latter we add a so called strut — a box with a certain height (or width) and no width (height). In LaTeX we have `\rule` for that, TeX needs a `\hrule` or `\vrule`.

```
\hbox{\vrule width\d\dDisk height1\Disk depth0\Disk}
\hrule width0\Disk height0.25\Disk depth0\Disk
\nointerlineskip
```

Another problem is that we want to center the disks that are on one pile. For this purpose we introduce a global variable that contains the size of the largest disk (it should be initialised by `\TowerOfHanoi`):

```
\newcount\maxsize
```

Then, by using another scratch variable, we can calculate how much whitespace we must `\kern` in front of the disk: `\maxsize-\d \Disk`. The entire disk print command is:

```
\c=\maxsize \advance\c by -\d
\hbox{\kern\c\Disk\vrule width\d\dDisk height1\Disk depth0\Disk}
\hrule width0\Disk height0.25\Disk depth0\Disk
\nointerlineskip
```

Furthermore, every pile must have the same width of `\maxsize\dDisk`. We can *not* accomplish this by adding a `\kern\c\Disk` to the right of each disk, since a pile may be empty. Again, we add a strut. One with width `\maxsize\dDisk`. Finally, we add a little whitespace to the left and right of a pile so that two consecutive piles will not bump. The entire macro is:

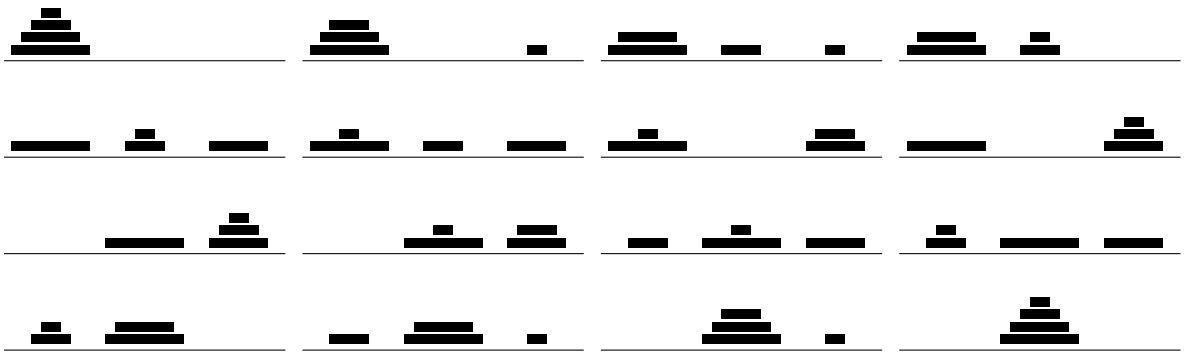
```
\def\Print#1{%
  \e=#1%
  \kern 1 mm
  \vbox{%
    \hrule width \maxsize\dDisk height0\Disk depth0\Disk
    \loop\ifnum\e>0
      \d=\e \divide\d by 8 \multiply\d by -8 \advance\d by \e
      \divide\e by 8
      \c=\maxsize \advance\c by -\d
      \hbox{\kern\c\Disk\vrule width\d\dDisk height1\Disk depth0\Disk}
      \hrule width0\Disk height0.25\Disk depth0\Disk
      \nointerlineskip
    \repeat}
  \kern 1 mm}
```

To make every set of three piles of equal height, we add another strut, this time in the `\PrintTowers` macro. Each disk has height 1, a space of 0.25 is added after each disk and a pile has a maximum of 7 disks. So a height of $(7 \times 1.25 = 8.75)$ 9 should work. Finally, to show which piles form one state, we underline them (which must be done in math mode):

```
\def\PrintTowers{%
  $\underline{
    \Print{\number\TowerOne}
    \Print{\number\TowerTwo}
    \Print{\number\TowerThree}
    \vrule width0\Disk height9\Disk depth0\Disk}
  $}
```

4.5 The kick

I will no longer delay your kick. Here it is, the result of `\TowersOfHanoi{4}`:



As you may remember: the height of a disk is `1\Disk` which should be equal to 1.3mm. Are they much thicker? Take a ruler and check before you complain.

The answer to the `\puzzle` is `ABCAB`. The first `\a` expands into `A\def\a{B...}` which redefines `\a`. From this example you can see that a control sequence `\b` needs not to be defined when it appears in the replacement text of a definition. The puzzle also shows that TEX doesn't expand a macro until it needs to.

The output for the second puzzle is `12345464` but the spacing is messy. Note the space in the definition of `\s` to delimit constants.

```
\newcount\c \newcount\d \newcount\e \newcount\n \newcount\maxsize
\newcount\TowerOne \newcount\TowerTwo \newcount\TowerThree
\newdimen\Disk \Disk=0.13 cm \newdimen\dDisk \dDisk=2\Disk

\def\Print#1{%
  \e=#1%
  \kern 1 mm
  \vbox{%
    \hrule width \maxsize\dDisk height0\Disk depth0\Disk
    \loop\ifnum\e>0
      \d=\e \divide\d by 8 \multiply\d by -8 \advance\d by \e
      \divide\e by 8
      \c=\maxsize \advance\c by -\d
      \hbox{\kern\c\Disk\vrule width\d\dDisk height1\Disk depth0\Disk}
      \hrule width0\Disk height0.25\Disk depth0\Disk
      \nointerlineskip
    \repeat}
  \kern 1 mm}

\def\PrintTowers{%
  $\underline{
    \Print{\number\TowerOne}
    \Print{\number\TowerTwo}
    \Print{\number\TowerThree}
    \vrule width0\Disk height9\Disk depth0\Disk}
  $}

\def\Move#1#2{%
  \d=#1 \divide\d by 8 \multiply\d by -8 \advance\d by #1%
  \divide#1 by 8
  \multiply#2 by 8 \advance#2 by \d}

\def\Hanoi#1#2#3{\ifnum\n>0
  \advance\n by -1
  \Hanoi{#1}{#3}{#2}
  \Move{#1}{#2}%
  \PrintTowers%
  \Hanoi{#3}{#2}{#1}%
  \advance\n by +1
  \fi}

\def\TowersOfHanoi#1{% 0-7
  \TowerOne=0 \TowerTwo=0 \TowerThree=0 \n=#1%
  \loop\ifnum \n>0
    \multiply\TowerOne by 8
    \advance\TowerOne by \n
    \advance\n by -1
  \repeat
  \n=#1 \maxsize=#1 \PrintTowers
  \Hanoi{\TowerOne}{\TowerTwo}{\TowerThree}}

\noindent\TowersOfHanoi{4}
\bye
```