Excursions in Computing Science:
Week 11. Memory, Feedback and Automata

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

June 10, 2015

I. Prefatory Notes

1. Memory

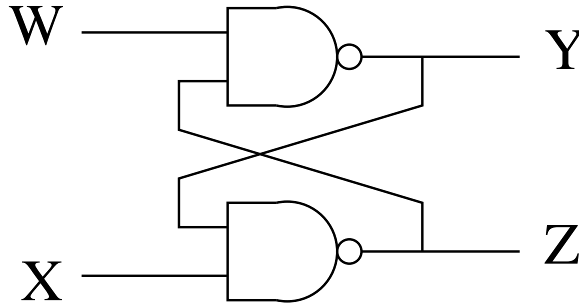
With boolean algebra we can make any logic gate, so in principle we can make any CPU circuit—adders, multipliers, etc.

The missing ingredient is *memory*.

This needs *feedback*.

The basic memory circuit is a *flipflop*.

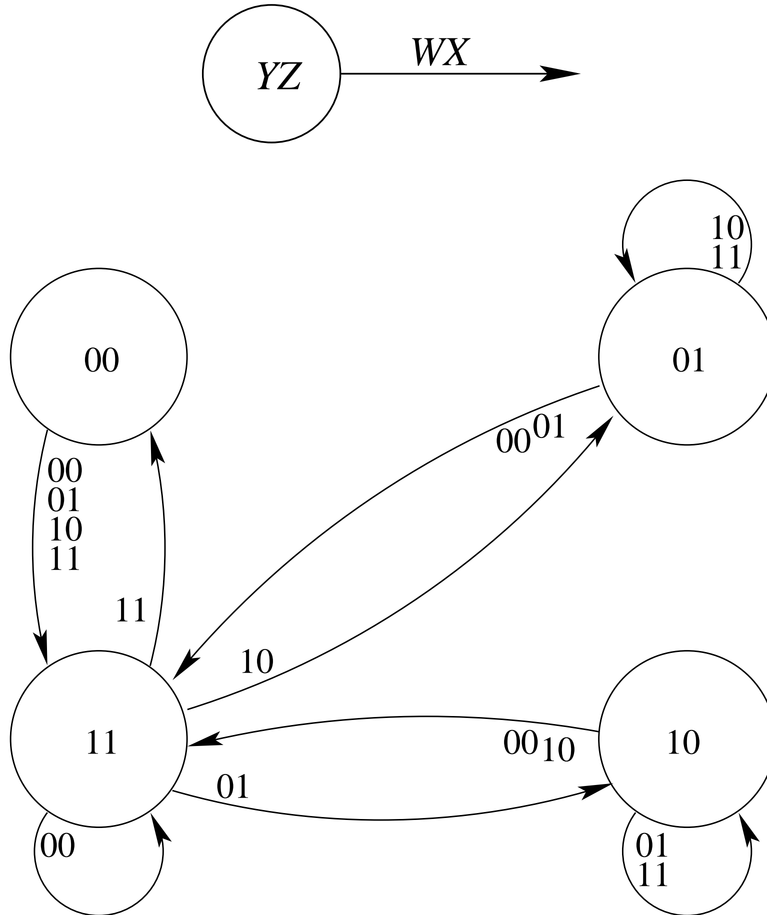
E.g. the **nand** flipflop.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
W_0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
X_0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Y_0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Z_0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
W_1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
X_1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Y_1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0
Z_1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0
	3	3	3	3	7	7	6	6	B	9	B	9	F	D	E	C

*Copyleft ©T. H. Merrett, 2006, 2009, 2013, 2015. 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 and their funding agencies.

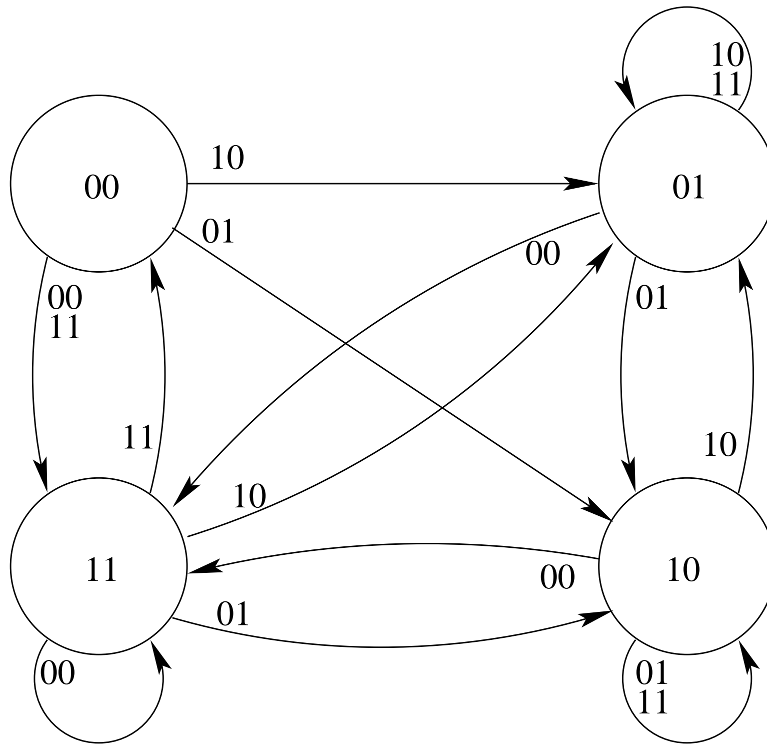
Let's consider YZ to be the state and WX to be the input.



Here it is in MATLAB. (Note that the outputs from a MATLAB function must be listed explicitly, and that the state, `y` and `z`, must be both input and output to be remembered for the next call to the flipflop function.)

```
function [w,x,y,z] = flipflopNandAll(w,x,y,z)
y1 = y;
z1 = z;
y = nand(w,z1);
z = nand(x,y1);
```

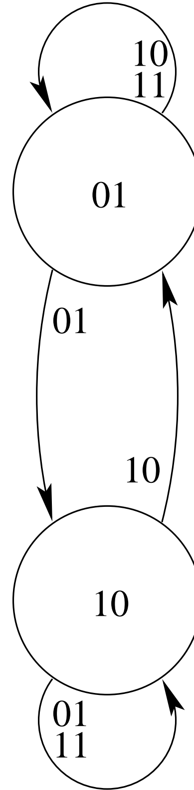
But let's follow these through right until the states stop changing.



Since the paths to stable states have at most two iterations (except for the $C \leftrightarrow F$ cycle), we can settle the flipflop easily in MATLAB.

```
function [w,x,y,z] = flipflopSettle(w,x,y,z)
[w,x,y,z] = flipflopNandAll(w,x,y,z);
[w,x,y,z] = flipflopNandAll(w,x,y,z);
```

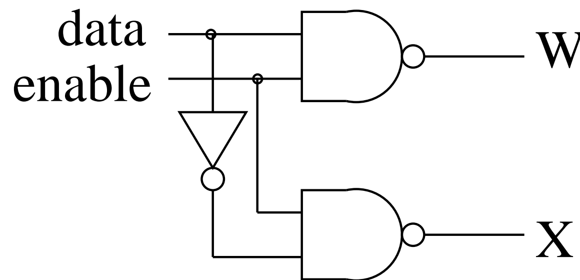
If we outlaw inputs $WX = 00$, states $YZ = 00$, $YZ = 11$ and keep only Y **xor** Z, we get the essential part of this figure.



So the flipflop can be in either of two states, which we can call 0 or 1 according to the value of Y:
0 is $YZ = 01$
1 is $YZ = 10$

That is, if we want it to remember 0, use $WX = 10$, then set $WX = 11$.
Or, if we want it to remember 1, use $WX = 01$, then set $WX = 11$.

Let's make this clear and avoid illegal states by adding a front end of two **nand** gates.



data	*enable*	*W*	*X*
0	0	1	1
0	1	1	0
1	0	1	1
1	1	0	1

Now the flipflop just remembers the value *data* had when *enable* was set to 1. What it remembers is always available on Y (the output from the flipflop on the first page, whose input is W and X).

Here it is in MATLAB.

```
% nand flipflop prefaced by nand front end
function [data,enable,y,z] = flipflopNandFEall(data,enable,y,z)
w = nand(data,enable);
x = nand(not(data),enable);
[w,x,y,z] = flipflopSettle(w,x,y,z);
```

Now we can make Read and Write functions for the flipflop (subject to MATLAB's restrictions on remembering the state, y).

```
function y = flipflopWrite(data,y)
[data,enable,y,z] = flipflopNandFEall(data,1,y,not(y));
```

```
function y = flipflopRead(data,y)
[data,enable,y,z] = flipflopNandFEall(data,0,y,not(y));
```

Boolean circuits without feedback are called "combinational"; with feedback they are "sequential".

Moral: memory needs feedback.

2. Control

Feedback is used heavily in control systems. Let's build a thermostatic furnace control.

The thermostat turns the heat on when the room falls below a certain temperature, T_C, and off when it rises above a higher temperature, T_H. We'll use two bits for temperature, t_H and t_C.

t_H	t_C	
0	0	$T < T_C$
0	1	$T_C \le T \le T_H$
1	1	$T_H < T$

Notice that $(t_H, t_C) = (1, 0)$ is excluded.

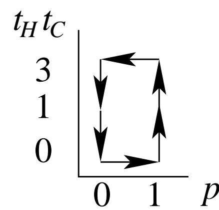
Another bit tells us if the heat is on or off.

p	
0	furnace off
1	furnace on

Let's work out the before and after pictures. We'll suppose it's cold outside.

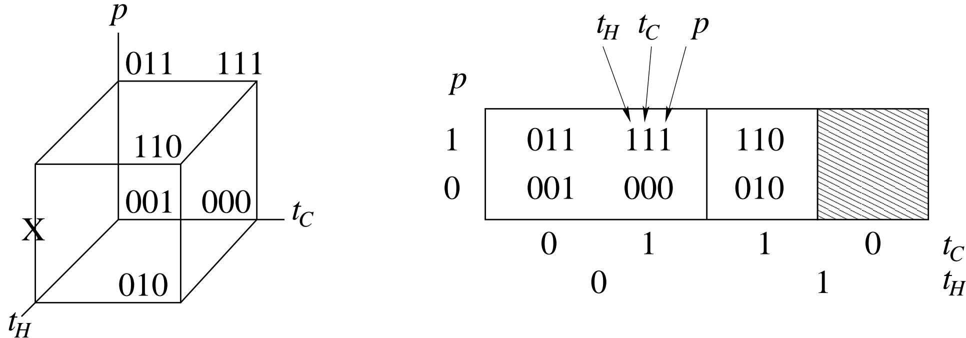
	before			after			
	t_H	t_C	p	t_H	t_C	p	
0	0	0	0	0	0	1	1
1	0	0	1	0	1	1	3
2	0	1	0	0	0	0	0
3	0	1	1	1	1	1	7
illegal	1	0	0		don't		
illegal	1	0	1		care		
6	1	1	0	0	1	0	2
7	1	1	1	1	1	0	6

t_H	t_C		
1	1	6	7
0	1	2	3
0	0	0	1
	p	0	1



- This is a *cycle*
- It is a *hysteresis* cycle: the system has *memory* of previous state(s) and does not return the way it came.

We can find the boolean equations for the furnace by using either a cube or a three-dimensional Karnaugh map.



$$
\begin{aligned}
p &\leftarrow pt'_H + t'_C t'_H \\
t_C &\leftarrow pt'_H + t_C t_H \\
t_H &\leftarrow pt_C
\end{aligned}
$$

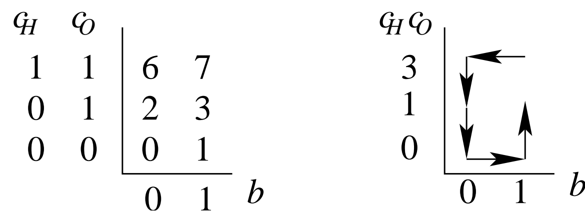
Note that these assignments are intended to give the same results no matter which order they are written in. So they will have to be modified in a program by assigning to temporary variables which are afterwards assigned to the left-hand sides.

3. Let's have some fun with a similar control system, an electric blanket control. Now we'll use two bits to give not temperature but comfort.

c_H	c_O	
0	0	comfort cold
0	1	comfort OK
1	1	comfort hot

We'll suppose that once you are comfortable, you leave the blanket on: $b = 1$.

	before			after			
	c_H	c_O	b	c_H	c_O	b	
0	0	0	0	0	0	1	1
1	0	0	1	0	1	1	3
2	0	1	0	0	0	0	0
3	0	1	1	0	1	1	3
6	1	1	0	0	1	0	2
7	1	1	1	1	1	0	6

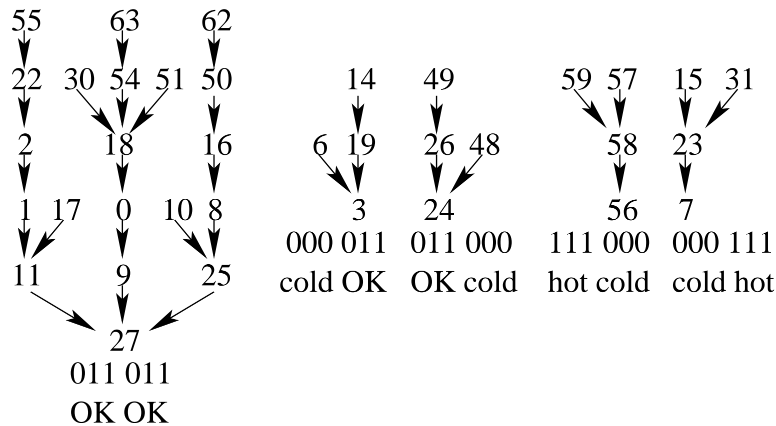


The boolean equations are

$$
\begin{aligned}
b &\leftarrow bc'_H + b'c'_O \\
c_O &\leftarrow bc'_H + c_O c_H \\
c_H &\leftarrow bc_H
\end{aligned}
$$

Now let's think about a double-bed electric banket with two controls, one for each side. *But* the couple who bought the blanket put it on the bed with the controls mixed up: she got his control and he got hers. What will happen?

There are thirty-six states, which flow as follows.



Note that this contains examples of *positive feedback* leading to *instability*: if our model did not limit the temperature to three values, one spouse could get colder and colder and the other hotter and hotter.

To work out and simplify the boolean equations, we need six dimensions: it is better to use a Karnaugh map.

c_{H_2}	c_{O_2}	b_2	$b_1=0, c_{O_1}=0, c_{H_1}=0$	$b_1=1, c_{O_1}=0, c_{H_1}=0$	$b_1=1, c_{O_1}=1, c_{H_1}=0$	$b_1=0, c_{O_1}=1, c_{H_1}=0$	$b_1=0, c_{O_1}=1, c_{H_1}=1$	$b_1=1, c_{O_1}=1, c_{H_1}=1$	$b_1=1, c_{O_1}=0, c_{H_1}=1$	$b_1=0, c_{O_1}=0, c_{H_1}=1$
1	0	0								
1	0	1								
1	1	1	7 → 7	15 → 23	23 → 7	31 → 23	55 → 22	63 → 54		
1	1	0	6 → 3	14 → 19	22 → 2	30 → 18	54 → 18	62 → 50		
0	1	0	2 → 1	10 → 25	18 → 0	26 → 24	50 → 16	58 → 56		
0	1	1	3 → 3	11 → 27	19 → 3	27 → 27	51 → 18	59 → 58		
0	0	1	1 → 11	9 → 27	17 → 11	25 → 27	49 → 26	57 → 58		
0	0	0	0 → 9	8 → 25	16 → 8	24 → 24	48 → 24	56 → 56		

These two control systems are "on-off", or binary, and so can be modelled with boolean circuits. Even though the temperature and the comfort each have three levels, we can model them with a

pair of booleans: four values, one forbidden.

Control systems are usually continuous and modelled by ODE, ordinary differential equations, or by finite difference equations if the continuum is discretized. (The independent variable is usually time.) Control theory is primarily concerned with *stability*—people don't become arbitrarily hot or cold—and *performance*—the system settles quickly to the desired goal. Control systems use feedback to these ends.

4. Biology

Living systems use feedback extensively, often of the continuous sort—try picking up your pen with your eyes closed!

A question of concern to a software person is: why aren't biological systems *brittle*, like programs? A misplaced comma can change a program run drastically, or even make it crash. (Language, too, being also a digital system, has this problem: "eats, shoots and leaves".) Organisms, on the other hand, *adapt*.

A question of interest to biologists is: since all organisms are at conception a single cell, how do later cells *differentiate* into skin, bone, nerve, optic, muscle, blood, kidney, .., and germ cells?

Maybe boolean circuits can help. Cell differentiation is a result of the same genetic code being *expressed* in different ways under different circumstances.

Let's look at a very simple system of gene expression, which won't take us all the way to cell differentiation or to the adaptability of organisms, but may give us a glimpse of how these much more complex systems might work.

The *lac operon* of *E. coli* is well understood by molecular biologists and we can make a boolean model of it. (Biologists call computer models "in silico" biology (in silicon) as opposed to "in vitro" (in glassware in the laboratory) or "in vivo" (in the living organism).)

There is a lot of biochemistry but it boils down to this:

- If lactose levels are high in the cell, the lac operon is *on* and the cell generates two enzymes: Z metabolizes the lactose; Y makes the cell wall permeable to let more in. The net effect is to let the bacterium extract energy from the lactose, and thus lower the lactose level in the cell.
- If lactose levels in the cell are low, the enzyme generation is inhibited.

We use three boolean variables with the following meanings (when valued 1):

e lactose is present in the environment
c lactose is present in the cell
l lac operon is on

	before			after			
	e	c	l	e	c	l	
0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
2	0	1	0	0	1	1	3
3	0	1	1	0	0	1	1
4	1	0	0	1	1	0	6
5	1	0	1	1	0	0	4
6	1	1	0	1	1	1	7
7	1	1	1	1	0	1	5

The cell differentiates into two distinct behaviours, depending on the environment:

no environmental lactose	0 is attractor	$2 \to 3 \to 1 \to 0$
environment has lactose	hysteresis loop	$4 \to 6 \to 7 \to 5 \to 4 \to \cdots$

This "cell differentiation" is not permanent—a change in the environment will swap it from one state to the other—but the process *is* genetic:

- The operon contains *lacY* and *lacZ* genes which code for proteins that open the cell walls (lactose permease) and metabolize the lactose (beta-galactosidase), respectively.
- The genes of the lac operon are transcribed (to RNA and thence to make the proteins) by the gene expression mechanism only if a "lac repressor" protein (coded by a *lacI* gene) is not bound to the operon's "promoter" site.
- When lactose is present, allolactose, a protein with levels correlated with the lactose level, binds to the repressor protein, effectively pulling it off the promoter.

Roger Miesfeld has some pictures of all these molecules [Mie01].

Since this is a genetic process, we can imagine that similar processes might actually change the cell permanently. Then we have differentiation.

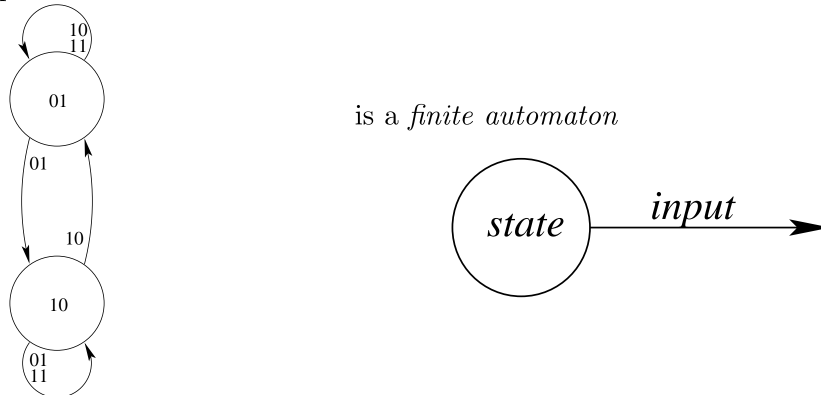
This simple system illustrates the idea of a "tipping point": it has one behaviour when lactose is low in the environment, but when the environmental lactose increases past a certain level, it abruptly changes behaviour.

States such as 0 in this example are *attractors*: the system can get into such a state and not get out of it. The cycle $4 \to 6 \to 7 \to 5 \to 4 \to \cdots$ is also an attractor: the system stays in the cycle. (Chaos theory is about "strange" attractors, which have so many states that their behaviour once in the attractor seems very hard to predict. Chaos mainly arises in continuous systems (of nonlinear differential equations) but it can be seen in boolean systems, too.

Finally, we can come back to the question of biological suppleness as opposed to software brittleness. The ability of a system to settle into an attractor (whether it be a single state or a cycle) from a variety of starting points is called "homeostasis". Usually the equivalent of changing a comma in a biological system just changes the initial state to another which eventually settles to the same attractor, so the organism or species does not crash.

5. Language

Back to the flipflop.



It changes state depending on the input.

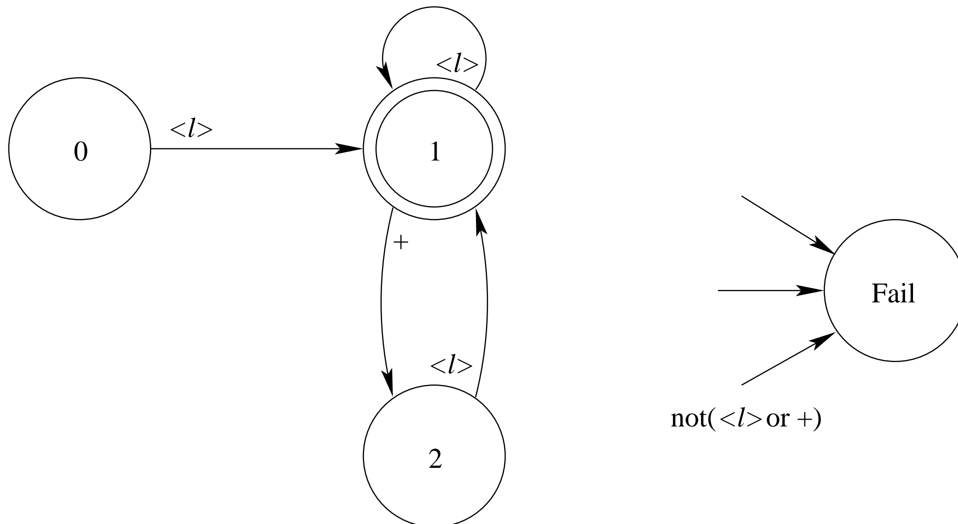
We can use an automaton to recognize the grammar of a language.

Let's check if segments of boolean algebra are expressions.

$ab + cd$	is
$ab+$	is not
$+a$	is not

Clearly a mechanism to recognize that a sequence of symbols does or does not satisfy a grammar, must have a *memory* of preceding symbols in the sequence. The devices with state we've been looking at are suggestive.

Suppose the allowed variables in our boolean language are any single letter, which we indicate with $< l >$. The **and** operator is implicit, given by adjacency of two variables (letters). The **or** operator is the explicit $+$.



In this figure, the doubly-circled state, 1, is an *acceptor* state:seebelow.

E.g., $ab + cd$

input	*state*	
$ab + cd$	0	
$b + cd$	1	
$+cd$	1	
cd	2	
d	1	
	1	acceptor state: OK

E.g., $ab+$

input	*state*	
$ab+$	0	
$b+$	1	
$+$	1	
	2	not acceptor state: not OK

This finite automaton can also be represented as a matrix

	input:	other	$< l >$	$+$
statebefore		0	1	2
init	0	3	1	3
accept	1	3	1	2
	2	3	1	3
fail	3	3	3	3
			stateafter	

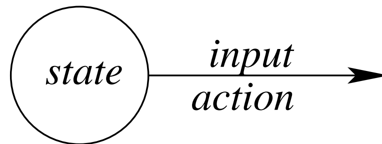
or as a set of *triples* $(0, < l >, 1), (1, < l >, 1), \cdots$

statebefore	0	0	0	1	1	1	2	2	2	3	3	3
input	0	1	2	0	1	2	0	1	2	0	1	2
stateafter	3	1	3	3	1	2	3	1	3	3	3	3

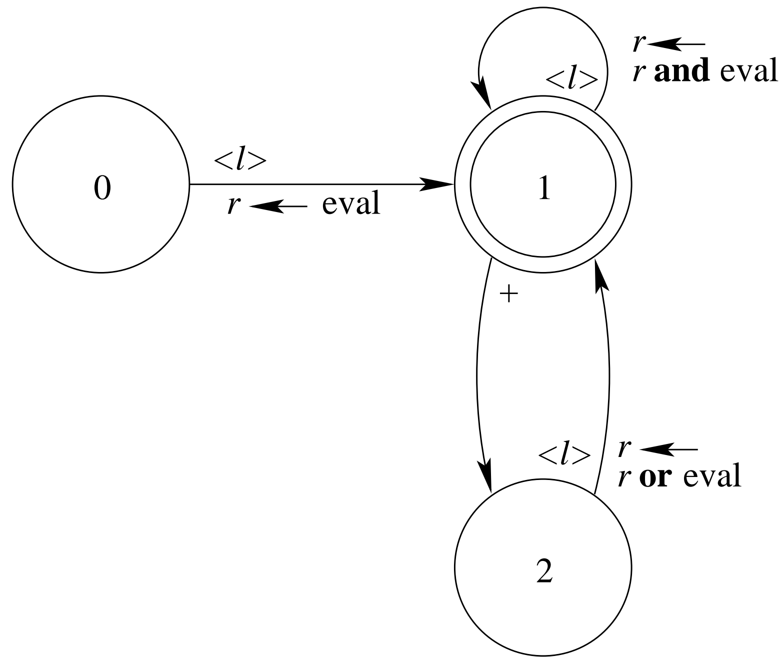
Here is a MATLAB version.

```
% triples[1+statebefore,1+plus|letter|other] = stateafter
% plus = 2; letter = 1; other = 0;
function ok = booleanFA(expression)
triples = [3,1,3;       %plus    3 2 3 3
           3,1,2;       %letter  1 1 1 3
           3,1,3;       %other   3 3 3 3
           3,3,3        %  state 0 1 2 3
        ];
state = 0;
for pos = 1:length(expression)
  if expression(pos)=='+', input = 2;
  elseif isletter(expression(pos)), input = 1;
  else input = 0; end
  state = triples(1+state,1+input);
end
ok = state==1;
```

6. Of course, it is more interesting to *evaluate* a (correct) expression than just to say it is correct. Let's add an *action* to each transition



and a *register*, r



a	**t**
b	**t**
c	**t**
d	**f**

input	*state*	r
$ab + cd$	0	
$b + cd$	1	**t**
$+cd$	1	**t**
cd	2	**t**
d	1	**t**
	1	**f**

Is this the result you expected? Operators got evaluated from left to right, not with the usual *precedence*.

In MATLAB:

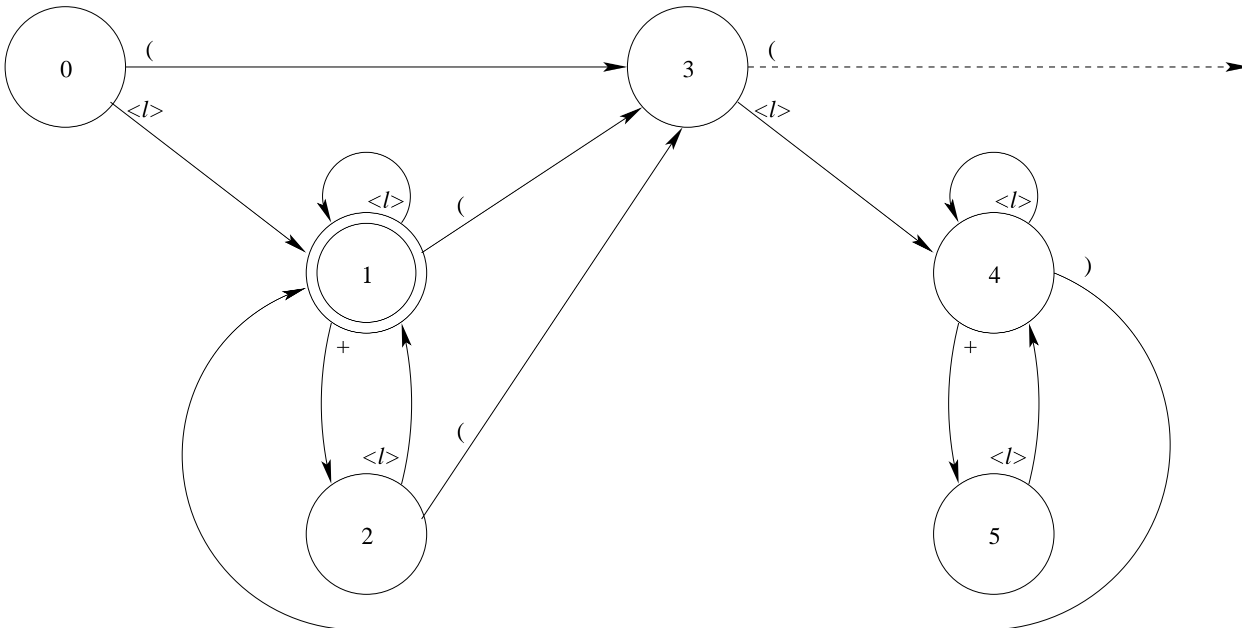
```
% triples[1+statebefore,1+plus|letter|other] = stateafter
% actions[1+statebefore,1+plus|letter|other] = eval|and|or|nop
% plus = 2; letter = 1; other = 0;
% eval = 'e'; and = 'a'; or = 'o'; nop = 'n';
function value = booleanFAeval(expression,identifiers,values)
triples = [3,1,3;       %plus    3 2 3 3
           3,1,2;       %letter  1 1 1 3
           3,1,3;       %other   3 3 3 3
           3,3,3        %  state 0 1 2 3
         ];
actions = ['nen';       %plus    n n n n
           'nan';       %letter  e a o n
           'non';       %other   n n n n
           'nnn';       %  state 0 1 2 3
         ];
state = 0;
for pos = 1:length(expression)
  current = expression(pos);
  if current=='+', input = 2;
  elseif isletter(current)
```

```
    input = 1;
    currval = lookup(current,identifiers,values);
  else input = 0; end
  action = actions(1+state,1+input);
  if action=='e', value = currval;
  elseif action=='a', value = and(value,currval);
  elseif action=='o', value = or(value,currval);
  end
  state = triples(1+state,1+input);
end
if state~=1, value = -1; end
```

7. So let's put in *parentheses*
e.g., $(ab) + (cd)$
and see if we can build an automaton to recognize the grammar for this.

$(ab) + cd)$	is not
$(ab + (cd)$	is not

If there is only one set of parentheses at a time, we can extend the finite automaton.



But if *nested parentheses* were allowed, e.g., $(a(b + c) + d)e$, then the finite automaton would fail. We'd need new extra pieces, starting at the dashed arrow transition from state 3, one piece for each allowed level of nesting.

That means an arbitrary number of extra sets of states and transitions. This is no good.

We need a more powerful automaton. Let's add a *stack*.

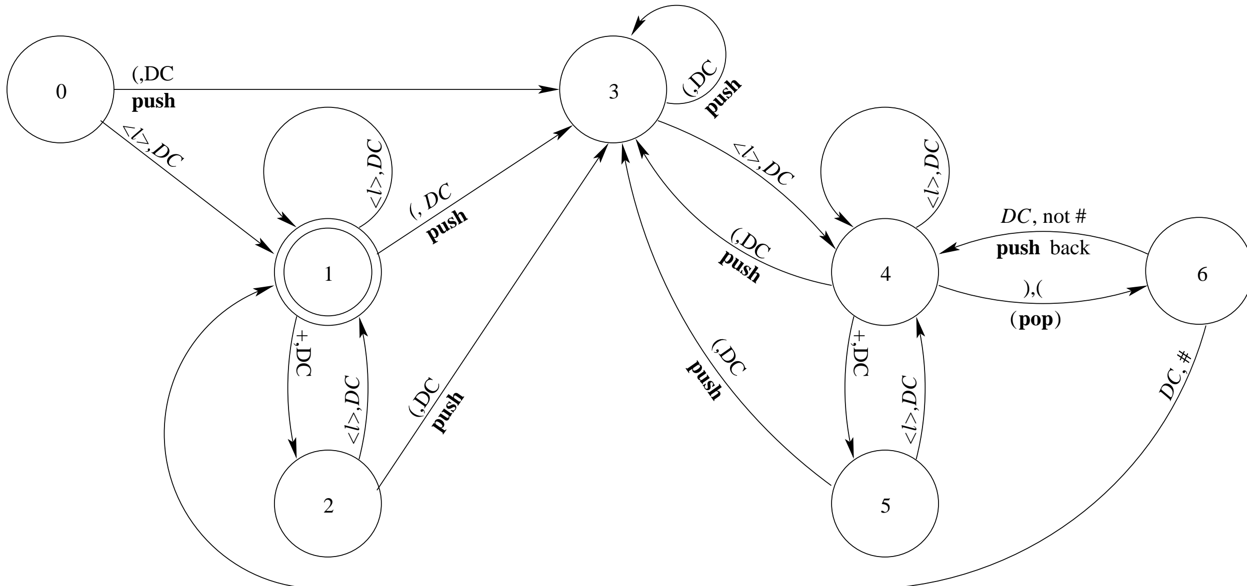
A stack is additional memory with last-in-first-out (LIFO) access. It has operations **push** with an argument (to put the argument on the top of the stack) and **pop** (to *take away* what's on the top of the stack).

The transitions of the new automaton have two labels and sometimes a **push** action:
input, stacktop(, **push**).

stacktop is the value on the top of the stack, and reading it is accompanied by a **pop**.

We'll assume one more frill: the stack always holds a bottom-of-stack symbol, say #, which remains even after **pop**.

Now we can build a *pushdown automaton* to recognize boolean expressions with parentheses.



$\mathcal{DC}$ means "don't care": don't look at the stack or the input, respectively, depending on which of the two positions the $\mathcal{DC}$ appears in.

input	*state*	*stack*	
$((ab)+(cd))$	0	#	
$(ab)+(cd))$	3	(#	
$ab)+(cd))$	3	((#	
$b)+(cd))$	4	((#	
$)+(cd))$	4	((#	
$+(cd))$	6	(#	
$+(cd))$	4	(#	
$(cd))$	5	(#	
$cd))$	3	((#	
$d))$	4	((#	
$))$	4	((#	
$)$	6	(#	
$)$	4	(#	
	6	#	
	4	#	
	1	#	accept!

8. To *evaluate* parenthesized boolean expressions we could add further actions to the pushdown automaton, as we did for the finite automaton earlier.

But there's a more direct way. we convert it to *reverse Polish* form and use a stack directly.

1. $((a.b)+(c.d)) \rightarrow ab.cd.+$ (The **and** operator is now explicitly given as ".".)

2.

a	**t**
b	**t**
c	**t**
d	**f**

input	*stack*	
$ab.cd.+$	#	
$b.cd.+$	t#	value of a
$.cd.+$	tt#	value of b, a
$cd.+$	t#	value of $a.b$
$d.+$	tt#	value of $c, a.b$
$.+$	ttt#	value of $d, c, a.b$
$+$	tt#	value of $c.d, a.b$
	t#	value of $(a.b) + (c.d)$

Here's the reverse Polish evaluation in MATLAB. The internal stack is implemented as an array and needs `push()` and `pop()` functions.

```
% letter: evaluate and push onto stack; operator: pop stack twice, push result
% e.g. of identifiers = 'abcd'
% e.g. of values = [1,1,1,0]
% e.g. of use
% booleanRevPolEval('ab.cd.+',identifiers,values)
% e.g. of use with shuntingYard:
% booleanRevPolEval(shuntingYard('((a.b)+(c.d))'),identifiers,values)
function value = booleanRevPolEval(expression,identifiers,values)
stack = [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]; top = 1;
for pos = 1:length(expression)
  current = expression(pos);
  if current=='+'
    [x,stack,top] = pop(stack,top);
    if x<0 | 1<x
      value = -1;
      return
    end
    [y,stack,top] = pop(stack,top);
    if y<0 | 1<y
      value = -1;
      return
    end
    [stack,top] = push(or(x,y),stack,top);
  elseif current=='.'
    [x,stack,top] = pop(stack,top);
    if x<0 | 1<x
      value = -1;
      return
    end
    [y,stack,top] = pop(stack,top);
    if y<0 | 1<y
      value = -1;
      return
    end
    [stack,top] = push(and(x,y),stack,top);
  elseif isletter(current)
    currval = lookup(current,identifiers,values);
    [stack,top] = push(currval,stack,top);
  else
    value = -1;
```

```
    return
  end
end
[value,stack,top] = pop(stack,top);


function [stack,top] = push(value,stack,top)
if top >= length(stack)
  return
else
  stack(top) = value;
  top = top + 1;
end


function [value,stack,top] = pop(stack,top)
if top < 2
  value = -1
  return
else
  top = top - 1;
  value = stack(top);
  stack(top) = -1;
end
```

Dijkstra's "shunting algorithm" to get reverse Polish form from a parenthesized expression uses a stack and a *queue.*

A queue is a first-in-first-out (FIFO) memory which we add to at the end and take away from at the beginning. (In the shunting yard algorithm, we don't take away.)

Parentheses and operators go through the stack. Variables go directly to the output queue.

input	*stack*	*output queue*
$((a.b)+(c.d))$	$\#$	
$(a.b)+(c.d))$	$(\#$	
$a.b)+(c.d))$	$((\#$	
$.b)+(c.d))$	$((\#$	a
$b)+(c.d))$	$.((\#$	a
$)+(c.d))$	$.((\#$	ab
$+(c.d))$	$(\#$	$ab.$
$(c.d))$	$+(\#$	$ab.$
$c.d))$	$(+(\#$	$ab.$
$.d))$	$(+(\#$	$ab.c$
$d))$	$.(+(\#$	$ab.c$
$))$	$.(+(\#$	$ab.cd$
$)$	$+(\#$	$ab.cd.$
	$\#$	$ab.cd.+$

Note that if we don't have parentheses around the whole expression, we'll usually have to pretend that we do, to flush the stack at the end.

9. We have the beginning of a hierarchy of automata
   finite automata

pushdown automata

each more powerful than the last. Can we go further? Must we go further?

‘The expression is $((ab) + (cd)$’ said Pat.
‘You forgot)’ said Jan.

This obeys two rules.

1. ‘ is always matched by ’, with nesting allowed.
2. (is always matched by), with nesting allowed.

Clearly we could check this with two stacks, but do we want to go on adding stacks as the grammars get more complicated? We rejected a similar approach when we moved from finite automata to pushdown automata.

Let’s change strategy again and build a *Turing machine*. This is a finite automaton with no stacks and its input on a *tape*. Only now the Turing machine is allowed to *write* on its tape, *anywhere*, as well as read anywhere from the tape. Turing went on to prove that this is a *universal* computer: *any* Turing machine can exactly imitate any other. (This is certainly not true for finite or pushdown automata: we had to build a special one for each grammar.) He also showed that the Turing machine is so rich that some questions it raises cannot be answered. Whether a Turing machine, given some input, will ever stop, for instance, is undecidable: the *halting problem*.

By the way, we don’t need a Turing machine to recognize the ‘(’) grammar: an intermediate automaton, the *linear bounded automaton*, can do that. This is a Turing machine with restricted writing.

These four levels of automata give the *Chomsky hierarchy* of the languages that they recognize.

finite automata
pushdown automata
linear bounded automata
Turing machines

10. Summary

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

- Flipflops—feedback and memory.
- Control—feedback and memory (hysteresis).
- Gene expression—attractors and tipping points.
- Automata and languages
  - finite state automata
  - pushdown automata
  - Turing machines

II. The Excursions

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

1. Work out the behaviour of a feedback circuit identical to the **nand** flipflop but with the two **nand** gates in the feedback loops replaced by **nor** gates.
2. Sequential circuits are usually clocked. Look up the “jk flipflop” and write the full boolean table for its workings.

3. Look up David Wheeler's work [Den07] on hardware seizures and discuss how the idealization of flipflops we've been looking at falls short of a complete description and how actual flipflops can cause significant computer hardware glitches.
4. Can you construct an even simpler feedback loop than the "cold outside" control system, which still shows hysteresis?
5. Is the "cold outside" control system a reversible boolean function?
6. How does the system behaviour change if it is warm outside (but not hot): (t_H, t_C) does not spontaneously fall to (0,0)?
7. How does the system behaviour change if it is hot outside: (t_H, t_C) does not spontaneously drop below (1,1)?
8. How would we extend the furnace control system to take outside temperature into account?
9. Write a MATLAB program to show the cycle of the furnace control system: implement the boolean equations; pick a first state, (c_0, b_0), and set element (c_0, b_0) of a 3×2 matrix to zero; find the next state, (c_1, b_1), and set that element to 1; continue, incrementing the value you put in the matrix until you think the program should stop; display the matrix.
10. Write a MATLAB program that will display the paths for any of the three heat pump systems (cold, warm or hot outside) as arrows in a matrix.
11. Show that the boolean equations for the single electric blanket system are as given. (Note: you can make free with the values in the forbidden zone.)
12. Is the single blanket control system a reversible boolean function?
13. Work out the flow paths in the double blanket example. Note the symmetry.
14. What is the probability that both spouses will wind up comfortable?
15. What is the probability that they will both have a bad night (say, both wake up to adjust their controls at least three times).
16. What prevents the couple from arriving at the hot-OK or OK-hot states?
17. On the Karnaugh map of the double blanket system, convert the numbers to binary and work out the boolean equations for the system.
18. On the Karnaugh map of the double blanket system, trace the routes of one of the longest paths and of the path that is symmetric to it.
19. What happens in the double blanket case if the controls are *not* mixed up? What is the longest path ending with both spouses comfortable?
20. Find a way to model a single electric blanket control with five settings.
21. When you pursue a butterfly you need feedback to tell you how close you are. Does this apply also to the pursuit of happiness? What is the effect on your happiness of asking how close you are to happiness? Is it true that one can easily make oneself unhappy but needs others to be happy?Is it inevitable that the pursuit of happiness leads to beating up on others?
22. "Attractors" are discussed in Note 4. Which of the following qualities could be described as attractors: addiction, asymmetry of time, earworm, eros (romantic love), free will, ideaworm (ideology), individuality, me-you-ness (consciousness, sentience, ..), political power?

23. It is unsatisfactory to characterize humans as self-aware because any feedback system is self-aware in a sense. It must know the difference between its current state and the state it must be in to achieve its goal. What about characterizing humans as “other-aware”? What mechanism would permit other-awareness? Hint: look up John von Neumann (1903–1957) and Edgar Frank Codd (1923–2003) for their work on self-reproducing automata.

24. Look up Theodore E. Djaferis’ *Automatic Control* [Dja00]: this introduces the differential equation side of control theory. How does he prevent a robot car from crashing into the wall?

25. Look up the Belousov-Zhabotinsky reaction and model it using boolean circuits.

26. Find all the attractors in all the feedback examples discussed in the notes.

27. a) Design the Boolean circuit for a “latch” with a state, s, and an external stimulus, e: the latch is initially off ($s = 0$) until it is flipped on ($s = 1$) by a stimulus ($e = 1$); thereafter it remains on no matter what happens to e. Note: this is an irreversible operation.
    b) Modify the latch to a “dead-man” latch, which is flipped on by the external stimulus being turned off ($e = 0$). (The operator holds down the button to keep the latch off, but if he lets go, the latch goes on.)
    c) Adapting the above ideas, build a “piggy bank”, which is intact ($s = 1$) until it is hit ($e = 1$), and remains broken ($s = 0$) thereafter, independently of the value of e.
    d) Build a “learning latch”, which takes two tries ($e = 1$ twice) before staying latched ($s = 1$). How about three tries?

28. Try combining the latch with the lac operon so that once the operon gets into one of its two states, it stays that way, independently of the environment.

29. Look up Malcolm Gladwell’s *The Tipping Point* [Gla00]: try to model his examples with boolean circuits. For a start, model a glass full of water which gets knocked on its side: a slight knock does not move its centre of mass further than its outside edge, so it jostles but winds up upright again; a hard knock does push it over and the water spills out.

30. Gene expression such as the lac operon of Note 4 is frequently discovered by genomicists by using sequences of *microarrays*. A microarray records the state of the cell at any one time by capturing the various products expressed by some of the genes. A sequence of them is a *timeseries*.
    We can illustrate this for the lac operon by a sequence of vectors, such as

    $$\begin{pmatrix} e \\ c \\ l \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & \\ 0 & 1 & 1 & 0 & 0 & .. \\ 0 & 0 & 1 & 1 & 0 & \end{pmatrix}$$

    This example captures the cycle in Note 4. It is likely that the other sequence, $2 \cdots 3 \cdots 1 \cdots 0 \cdots 0 \cdots$, would reach its attractor before getting recorded in the microarray.
    From this timeseries, we have only part of the before-after transition table of Note 4.

	before			after			
	e	c	l	e	c	l	
4	1	0	0	1	1	0	6
5	1	0	1	1	0	0	4
6	1	1	0	1	1	1	7
7	1	1	1	1	0	1	5

    Can we use this to figure out, at least partially, the boolean equations governing this gene expression?

A method due to [CCR11] uses *discrete Jacobians* to detect how each Boolean variable, e, c and l, affects the other.
The idea is to move the "before" variables from one state to each of its neighbour states and see what this does to the "after" variables. For example the neigbour states of `00` are `10` and `01`, and the neighbour states of `110` are `010`, `100` and `111`. Plainly if we have v variables, any one state will have v neighbours, and the record of the influences of each of v variables on each of the others will be a $v \times v$ matrix, which we call the Jacobian.
(You will also have encountered Jacobians for continuous functions in Book 9c, based on slopes, which can serve the same purpose.)
To do this thoroughly, we need a complete set of neigbours. But with the partial data we might get from a timeseries we do not always have complete sets of neighbours. For instance, in the lac operon loop above we do not have all three neighbours of any of the "before" states. So let's see what we can do with the second and third variables only, c and l.
Here, for $e = 1$, we have all four possibilities for c and l, `00, 01, 10` and `11`. So each of these states has all its neighbours. Let's get the Jacobian matrices for each of these.

$$J(e00) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad J(e01) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$J(e10) = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \qquad J(e11) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Here's an explanation for $J(e10)$. When the before state (c, l) changes from (1,0) to (0,0) (the first of the variables, c, changed), the after state (c, l) changes from (1,1) to (1,0) (a change in the same variable). So c influences c. When the before state (c, l) changes from (1,0) to (1,1) (the second of the variables, l, changed), the after state changes from (1,1) to (0,1) (a change in the first of the variables again). So l influences c.
It seems evident that we want to combine all these discoveries about influence we've recorded in the Jacobians, so we take their union.

$$J(e00) \cup J(e01) \cup J(e10) \cup J(e11) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Thus we now know what functions of the before-variables the after-variables will be. Let's call these functions $f_c()$ and $f_l()$. Their dependences can be read off the columns of this J-matrix: $f_c() = f_c(c, l)$ and $f_l() = f_l(l)$.
Now looking back to the before-after table enables us to specify these functions as far as we can with only the two variables.

$$\begin{aligned} f_c(c, l) &= [00 \to 1, 01 \to 0, 10 \to 1, 11 \to 0] \\ f_l(c) &= [0 \to 0, 1 \to 1] \end{aligned}$$

This becomes

$$\begin{aligned} f_c(c, l) &= l' \\ f_l(c) &= c \end{aligned}$$

a) Using a Karnaugh map (Note 5 of Week 10) and the full table in Note 4 work out the full Boolean equations for the lac operon. Compare this to the incomplete results above. (Because the variable e was left out of the Jacobians, but is always 1, does it make sense to **and** each of the above with itself, e?
b) Look up [CCR11] for an example of solution with 4×4 Jacobians.
c) Can you get better results with the other examples in these Notes, e.g., the furnace in Note 2 or the blanket in Note 3?
d) Since the neighbours of states `00, 01, 10` and `11` overlap so much, can we avoid looking at all four Jacobians?

31. Look up Stuart A. Kauffman's *The Origins of Order* [Kau93]: look for boolean circuits in biology; look for chaos in boolean circuits; why do biologists cite Alan Turing?

32. Look up James Gleick's *Chaos* [Gle88]: how does chaos relate to fractal geometry?

33. Look up "homeostasis" and think about what its presence in a software system might mean for a) brittleness and b) programmability.

34. The "(great) ocean conveyor" or "thermohaline" circulation that gives rise to the Gulf Stream is a feedback system. Look it up! Model it! Can you get a tipping point?

35. *The rich get rich and the poor get children*
Ain't we got fun
The rich get rich and the poor get laid off
Ain't we got fun
—Gus Kahn, Raymond B. Egan and Richard Whiting, 1921

A poor village family ([Sac05, pp.196–200, 227–38]) might be modelled in terms of the following boolean variables: g, *is cultivating a garden*; f, *buys/has fertilizer* (diammonium phosphate, about $25/bag); a, *buys/has artesunate* (an anti-malarial drug costing about $2 per treatment); s, *gets sick with malaria*; and m, *earns money*.

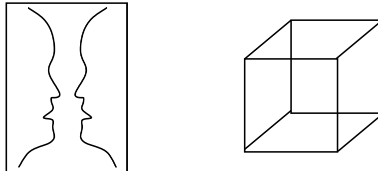
Possible transitions include: earning money enables one to buy fertilizer or artesunate; gardening earns money; having fertilizer permits gardening; having malaria and artesunate leads to recovery from the malaria. Show that the full transition table, with $gfasm$ expressed as integers, is

from	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
to	0	8	2	6	4	12	0	1	16	17	10	14	20	21	8	9

from	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
to	1	9	3	7	5	13	1	9	17	25	11	14	21	29	9	13

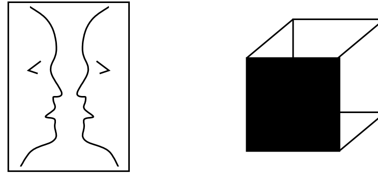
What are the attractors of this system? Can you find a "poverty trap" such as Sachs mentions? Enhance the model to take into account two levels of gardening: a low level permits subsistence but not income; a higher level permits both. Build in the possibility of capital investment resulting from higher income, for example in mosquito nets to make malaria "impossible" and artesunate unnecessary. What about a component for community investments, such as a water tap closer to the village, a generator to charge batteries used to give light after dark for study, or a truck to transport produce or patients?

36. Figures such as the faces/candlestick and the cube shown are ambiguous: as we gaze, the faces alternate with the candlestick and the cube alternates from a top view to a bottom view.



a) Model viewing one of these as a system which requires some "strength" to hold on to our current interpretation. The "strength" fades, and when it drops to zero, we switch to the other interpretation, back at full strength. Use a boolean variable, f, which is `1` for the "faces" interpretation and `0` for the "candlesticks" interpretation (or, correspondingly, for the top and bottom views of the cube, respectively). Use two variables for three levels of strength,

`00, 01` and `11`.
b) If the figures are disambiguated, it becomes (almost) impossible to switch away from the favoured interpretation.



Extend your model with a variable, e, for "eyes" (or for shading one face of the cube), so that it behaves as before when e=`0`, but, when e=`1` the eyes are drawn in and it is impossible to see the candlestick.
c) These ideas are developed from [Min86, sect. 25.1]. In section 20.2, Minsky also addresses ambiguity in language: *John shot two bucks* could refer to betting ("shot" means "bet" in some U.S. slang) or to hunting. Context is needed to disambiguate, say John holding poker chips or John in camouflage gear. Would similar models apply? In section 16.4, Minsky shows flipflop-like diagrams to express the kind of "locking-in" and "locking-out" that may go on in our minds: explore these with boolean circuits.

37. **Traffic memory.** a) (Warmup) Two seconds is considered to be a safe following distance when driving a car. This is convenient because it is independent of speed and because it is easy to measure as you drive behind another car. At 100 kph (kilometers per hour), about 30 m/s, two seconds is 60 m behind the car ahead. At 50 kph, about 15 m/s, it puts you 30 m behind.
Two seconds is also a plausible reaction time, not just between seeing the brake lights go on and putting on your own brakes, but also the whole process of adjusting deceleration to match.
Show that if the car ahead suddenly decelerates to a slower speed and you match his deceleration after a two-second delay, then if you were initially driving two seconds behind, you will still be two seconds behind at the new speed. (Explore this for initial and final speeds of 30 m/s and 15 m/s and deceleration of say 3 m/s every second (i.e., 3 m/s^2): find the average speed difference over the first second (he is decelerating but you have not reacted yet), the second second (he is decelerating and you still have not reacted), the third second (you are both decelerating), and so on until the final two seconds when he has reached the final speed and is no longer decelerating but you are still slowing down to the final speed. Try it again for different rates of deceleration, say 2.5 m/s^2, or 1.5 m/s^2. Make a general argument for any rate of deceleration.)
What happens if your reaction time is shorter or longer than two seconds?
b) (Traffic memory) Show that an indefinitely long line of cars following each other will preserve a memory of a slowdown: think of the signal and slowdown being propagated backwards. At what speed does it propagate? If the cars are following each other at the reaction distance of two seconds, show that this speed is the same as the initial speed of the cars, so that the "memory" of the slowdown stays at exactly the same place on the road. What if the cars are closer together than two seconds? Further apart? Can we think of the reaction time varying, especially when the cars are further apart? Under what circumstances will the memory disappear?
c) Since there is a memory, where is the feedback?
d) While we are speaking of driving cars, how much longer does it take to brake to a complete stop from 100 kph than from 50 kph? Why?

38. Write a finite automaton which recognizes unparenthesized boolean expressions including **not**, '.

39. Design a translator from Roman to Arabic numerals.

40. The MATLABpak for this week contains `concretePoetry`, which I wrote to try to generate language like the following, sent by a friend.

    your news views spoken gently &
    with wisdom patience forgiveness
    are retold & your thoughts sought

    bridging distance effort spins our
    realities satellites crashing onto
    our moon & reflecting back pluto..

    there is good light to press shirts
    armour bright for colosseum game
    hostile friendly fire with not a speck
    of common point of view to be found

    same for you too probably so this
    just a little chickadee song of hope

    It generates results such as

    our light thoughts song too not a game there a hostile

    shirts with patience thoughts to game onto retold view

    friendly little chickadee light with good pluto speck our speck

    retold pluto onto your light light hope to chickadee onto realities

    a) Try running it.
    b) Rewrite the array `automaton` as triples

    (<part of speech>, <part of speech>, <probability>)

    showing the probability a transition is made from one part of speech (*adjective, adverb, article, noun, preposition, pronoun* or *verb*) to another. Note that *start* and *stop* are also included as "parts of speech" in `automaton`. Explain how I got the automaton to jump from state to state with the probabilities you have found.
    c) Rewrite your triples as a graph.
    d) Add extra edges to the graph showing the probabilities that any particular word is emitted when the automaton is in the state corresponding to its part of speech.
    e) What is the shortest possible output from `concretePoetry`? The longest? Modify the code to count the number of words output and run the program 1000 times putting the counts into a histogram: that is if 17 outputs have 5 words, the 5th entry of the array `histogram` should have the value 17. What do you think this histogram will look like? Plot it!
    f) Given the result,

    there is good light to press shirts armour bright for colosseum game

    what is the sequence of states (parts of speech) that must have given rise to it? What is the probability that this result will be generated?
    g) The last question is easy in this example because no one word occurs as more than one

part of speech. This is not always the case: "fire" and "view" could be verbs as well as nouns, "game" and "light" could be adjectives as well as nouns, and so on. Then inferring the most likely sequence of states behind an utterence and the probability that the given automaton will utter it is much more subtle. Look up "hidden Markov models" and the Viterbi algorithm. (I believe the Wikipedia algorithm for Viterbi is wrong: I get the sequence S,R,R and the probability 0.0471754. Work through the example yourself and see whom you agree with.)
h) Note that `automaton` does not use the conjunctions (`&, so`), because conjunctions may combine adjectives with each other, adverbs with each other, nouns with each other, .., and whole clauses with each other. Why would this be a problem? Thus `concretePoetry` cannot generate everything my friend wrote. Modify the "grammar" expressed by the automaton to include conjunctions of clauses (whole sentences: "`I hit the ball and I ran to first base`").
i) Rerite the triples, omitting triples containing `start`, as a matrix, and write the triples containing `start` as a vector (remember to include all parts of speech in this vector). Use MATLAB to find out if the matrix has a fixed point, by applying it repeatedly to the starting vector. How do you interpret the result? Look up [H02] for some background.
j) A simpler probabilistic automaton can compose nursery tunes based on the probabilities of going from one note to another in existing tunes. Look up [Pin56] and build his probability matrix into an automaton which plays "banal tunes". (You'll need to do some research into note frequencies and MATLAB's `sound()` capability to make a simple scale player.)

41. Write the pushdown automaton in the notes as a set of quintuples
    (*statebefore, input, stacktop, stateafter, action*)
    Use NOP (no operation), PUSH <argument>, or READTOP as actions. (READTOP reads the top of the stack without **pop**ping it.)

42. Extend the pushdown automaton to recognize parenthesized boolean expressions including **not**, '.

43. Design a two-stack automaton to recognize the '(') grammar.

44. Write the shunting yard algorithm as a MATLAB function, with supporting functions for the stack and the queue. Write it so that it can be used along with `booleanRevPolEval()` as shown in the notes, section 8.

45. Look up Edsger Wybe Dijkstra, 1930–2002. Why is the shunting yard algorithm so called? What is quicksort? What is a semaphore?

46. Look up Turing machines and build one to recognize the '(') grammar. Look up linear bounded automata and show that your Turing machine is one.

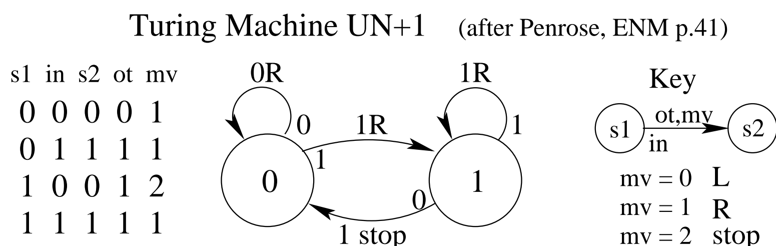
47. Here is a computation that never stops for any value of n:
    find an odd number that is the sum of n even numbers
    (Try it!).
    Here is a computation that stops only for $n = 0$, 1, 2 and 3:
    find a number that is not the sum of n square numbers
    (Check that it stops for $n = 0$, 1, 2 and 3. What are the answers in each case? Do not attempt to prove that it never stops for larger n—but this has been proven by the human mathematician, Lagrange: all numbers are the sum of four squares, and hence of five (just use 0), six, and so on.)
    These are examples of computations, depending on n, that never stop for at least some n. Imagine that we have a list of *all* different such computations, $C_0(n), C_2(n), C_2(n), C_3(n), ...$ Now imagine that there is a computation, A, depending on two numbers, q and n, which determines (proves) that $C_q(n)$ does not stop. $A(q, n)$ does this by calculating, say some sort

of mathematical induction (see Week 12), and stopping when, and only when it gets a result equivalent to asserting that $C_q(n)$ does not stop.
Now consider the "diagonal", $A(n,n)$. This is one of those computations that depends on n, so it must be one of the Cs, say $C_k(n)$. Do the diagonal thing again and consider $A(k,k)$. Now show that

if $C_k(k)$ stops then $C_k(k)$ does not stop.

This contradiction (see [Pen94, sect. 2.5] and the Excursion in Week 10 on diagonal proofs by contradiction) shows that no program can determine if any arbitrary program halts, a limitation on computation established by Alan Turing in 1937 (following similar limitations on logical reasoning established by Kurt Gödel in 1931). For an explicit Turing machine showing this contradiction, see Appendix A at the end of [Pen94]'s chapter 2.

48. A Turing Machine is an automaton with a potentially infinitely long "tape" from which it can read, and on which it can write, any number of 0s and 1s. As well as being able to change state depending on what input it reads from the tape, it also outputs a 0 or 1 to that position of the tape and it changes position by -1 ("left") or $+1$ ("right") for the next input. Here is the diagram, and its array representation, for a Turing Machine which copies a sequence of 1s back to the tape and then adds a final 1 in place of the next 0 it encounters. (It will ignore any number of leading 0s before it gets to the string of 1s.)



This Turing Machine [Pen89, p.41] actually adds 1 to a number, n, represented in "unary", that is, as a string of n 1s. Look up this reference and check that the state diagram and the array representation actually correspond to Penrose's notation for UN+1. Penrose also gives a Turing Machine which will double a number in unary representation, e.g., turning `...000111000...` (3) into `...000111111000...` (6).
Write MATLAB functions to
a) create the array representations of these two programs (e.g., `program = unaryIncrementTM()`),
b) create a tape with a given unary number embedded in a reasonable number (which you cannot unfortunately make infinite) of zeros before and after (e.g., `tape = makeTuringTape(tapelength, position,[1,1,1])`, where `position` gives the position of the first 1 of the data), and
c) execute your program on your tape (e.g., `[minpos,pos,maxpos,tape] = executeTuring(program,pos,tape`
where I suggest you use `minpos` and `maxpos` to track the extremes of the positions on the tape read by the Turing Machine).
d) For a physical implementation of a Turing macine, look up [NHVG09].

49. A few pages later, Penrose improves the unary representation to "extended binary notation", which modifies binary notation as follows.

`0`	$\to$	`0`
`1`	$\to$	`10`
,	$\to$	`110`

The advantage is that extended binary can represent a sequence of numbers on the tape, separated by the `110` (which we could think of as a comma), or a single number, terminated by `110` (remember that the tape has an indefinite number of 0s before and *after* the number

ends, so we must be able to tell where this end is). Ordinary numbers never have two consecutive 1s in extended binary, so the `110` delimiter is plainly identified.
Penrose goes on to give Turing Machines to increment and to double numbers represented in extended binary [Pen89, p.46].
a) Code these two Turing Machines and run them with `executeTuring()` from the previous excursion. Note that Penrose's XN×2 is wrong and you will have to correct it: draw the state diagram, try it out, and see what changes must be made.
b) Write functions to convert between ordinary and extended binary—or, for that matter, between decimal numbers and extended binary.

50. Look up Noam Chomsky. What is a transformational grammar?

51. Look up Kee Dewdney's *The Turing Omnibus* [Dew89] or *The New Turing Omnibus* [Dew93] (call them [61] and [66], respectively) chapters 2, 7, 14, 26, 31, 38, 44, 51, 59 and 66 [66] (2, 7, 13, 23, 28, 35, 41, 48, 55 and 60 [61]).

52. Any part of the Preliminary Notes that needs working through.

References

[CCR11] Ruth Charney, Jacques Cohen, and Aurelien Rizk. Efficient synthesis of a class of Boolean programs from I-O data: Application to genetic networks. *Discrete Applied Mathematics*, 159(6):410–9, March 2011.

[Den07] Peter J Denning. The choice uncertainty principle. *Comm. ACM*, 50(11):9–14, Nov. 2007.

[Dew89] A. K. Dewdney. *The Turing Omnibus: 61 Excursions in Computer Science*. Computer Science Press, Rockville, MD, 1989.

[Dew93] A. K. Dewdney. *The New Turing Omnibus: 66 Excursions in Computer Science*. Computer Science Press, Rockville, MD, 1993.

[Dja00] Theodore E. Djaferis. *Automatic Control: The Power of Feedback; using MATLAB*. Brooks/Cole, Pacific Grove CA, 2000.

[Gla00] Malcolm Gladwell. *The Tipping Point: How Little Things Can Make a Big Difference*. Little, Brown and Company, New York, 2000.

[Gle88] James Gleick. *Chaos: Making a New Science*. Penguin Books, New York, 1988.

[HÖ2] Olle Häggström. *Finite Markov Chains and Algorithmic Applications*. Cambridge University Press, Cambridge, U.K., 2002. London Mathematical Society Student Texts 52.

[Kau93] Stuart A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, New York, 1993.

[Mie01] Roger L. Miesfeld. E. coli hosts and plasmid biology. URL http://www.biochem.arizona.edu/classes/bioc471/pages/Lecture4/Lecture4.html, Sept. 2001.

[Min86] Marvin Minsky. *The Society of Mind*. Simon and Schuster, Inc., New York, 1986.

[NHVG09] Anders Nissen, Martin Have, Mikkel Vester, and Sean Geggie. The LEGO turing machine. URL http://www.youtube.com/watch?v=cYw2ewoO6c4, accessed 2010/11/24, 2009.

[Pen89] Roger Penrose. *The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics*. Oxford University Press, New York, 1989.

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

[Pin56] Richerd C. Pinkerton. Information theory and melody. *Scientific American*, 194(2):76–86, Feb. 1956.

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