Fundamentele Informatica 3
Antwoorden op geselecteerde opgaven uit Hoofdstuk 9

John Martin: Introduction to Languages and the Theory of Computation

Jetty Kleijn

Najaar 2008

9.1 See Figure 6.5: a TM accepting $\{ss \mid s \in \{a,b\}^*\}$. We give the configuration sequence of the computation for input *abaa*:
$(q_0, \underline{\Delta}aaba) \vdash (q_1, \Delta\underline{a}aba) \vdash (q_2, \Delta A\underline{a}ba) \vdash (q_2, \Delta Aa\underline{b}a) \vdash$
$(q_2, \Delta Aab\underline{a}) \vdash (q_2, \Delta Aaba\underline{\Delta}) \vdash (q_3, \Delta Aab\underline{a}) \vdash (q_4, \Delta Aa\underline{b}A) \vdash$
$(q_4, \Delta A\underline{a}bA) \vdash (q_4, \Delta \underline{A}abA) \vdash (q_1, \Delta A\underline{a}bA) \vdash (q_2, \Delta AA\underline{b}A) \vdash$
$(q_2, \Delta AAb\underline{A}) \vdash (q_3, \Delta AA\underline{b}A) \vdash (q_4, \Delta A\underline{A}BA) \vdash (q_1, \Delta AA\underline{B}A) \vdash$
— *abaa* is of even length —
$(q_5, \Delta A\underline{A}BA) \vdash (q_5, \Delta \underline{A}aBA) \vdash (q_5, \underline{\Delta}aaBA) \vdash (q_6, \Delta\underline{a}aBA) \vdash$
$(q_8, \Delta A\underline{a}BA) \vdash (q_8, \Delta Aa\underline{B}A) \vdash$ CRASH
or, in other words, $\vdash (h_r, \Delta Aa\underline{B}A)$.
abaa is not of the form *ss* and not accepted by the Turing machine.

9.2
a $(q_0, \underline{\Delta}ab) \vdash (q_1, \Delta\underline{a}b) \vdash (q_1, \Delta a\underline{b}) \vdash (q_1, \Delta ab\underline{\Delta}) \vdash$
$(q_2, \Delta a\underline{b}) \vdash (q_5, \Delta a\Delta\underline{\Delta}) \vdash (q_6, \Delta a\Delta b\underline{\Delta}) \vdash (q_7, \Delta a\Delta\underline{b}b) \vdash$
$(q_7, \Delta a\underline{\Delta}bb) \vdash (q_2, \Delta\underline{a}\Delta bb) \vdash (q_3, \Delta\Delta\underline{\Delta}bb) \vdash (q_4, \Delta\Delta a\underline{b}b) \vdash$
$(q_4, \Delta\Delta ab\underline{b}) \vdash (q_4, \Delta\Delta abb\underline{\Delta}) \vdash (q_7, \Delta\Delta ab\underline{b}a) \vdash (q_7, \Delta\Delta a\underline{b}ba) \vdash$
$(q_7, \Delta\Delta\underline{a}bba) \vdash (q_7, \Delta\underline{\Delta}abba) \vdash (q_2, \underline{\Delta}\Delta abba) \vdash (h_a, \Delta\underline{\Delta}abba)$

b $(q_0, \underline{\Delta}baa) \vdash (q_1, \Delta\underline{b}aa) \vdash^* (q_1, \Delta abb\underline{\Delta}) \vdash (q_2, \Delta ab\underline{b}) \vdash$
$(q_5, \Delta ab\Delta\underline{\Delta}) \vdash (q_6, \Delta ab\Delta b\underline{\Delta}) \vdash (q_7, \Delta ab\Delta\underline{b}b) \vdash (q_7, \Delta ab\underline{\Delta}bb) \vdash$
$(q_2, \Delta a\underline{b}\Delta bb) \vdash (q_5, \Delta a\Delta\underline{\Delta}bb) \vdash (q_6, \Delta a\Delta b\underline{b}b) \vdash^* (q_6, \Delta a\Delta bbb\underline{\Delta}) \vdash$
$(q_7, \Delta a\Delta bb\underline{b}b) \vdash^* (q_7, \Delta a\underline{\Delta}bbbb) \vdash (q_2, \Delta\underline{a}\Delta bbbb) \vdash (q_3, \Delta\Delta\underline{\Delta}bbbb) \vdash$
$(q_4, \Delta\Delta a\underline{b}bbb) \vdash^* (q_4, \Delta\Delta abbbb\underline{\Delta}) \vdash (q_7, \Delta\Delta abbb\underline{b}a) \vdash^* (q_7, \Delta\underline{\Delta}abbbba) \vdash$
$(q_2, \underline{\Delta}\Delta abbbba) \vdash (h_a, \Delta\underline{\Delta}abbbba)$

c After going to cell 1 and entering state q_1 the Turing machine moves to the right of (what still remains of) the input string (state q_2); it remembers and erases the last symbol (q_3 for a; q_5 for b) and moves it one cell to the right after which it proceeds (in q_4, resp. q_5) to the first empty cell to the right where it writes a, resp. b. Then it moves back to the left (in q_7) until it encounters an empty cell to the left of which the last symbol of the rest of the input is waiting. The process repeats from q_2 until no input is left, after which the Turing machine goes from q_2 to the accepting state h_a.
We conclude that $(q_0, \underline{\Delta} w) \vdash^* (h_a, \Delta\underline{\Delta} x x^r)$ for all $x \in \{a, b\}^*$: the machine adds the mirror image to a word (and the input is shifted one cell to the right).

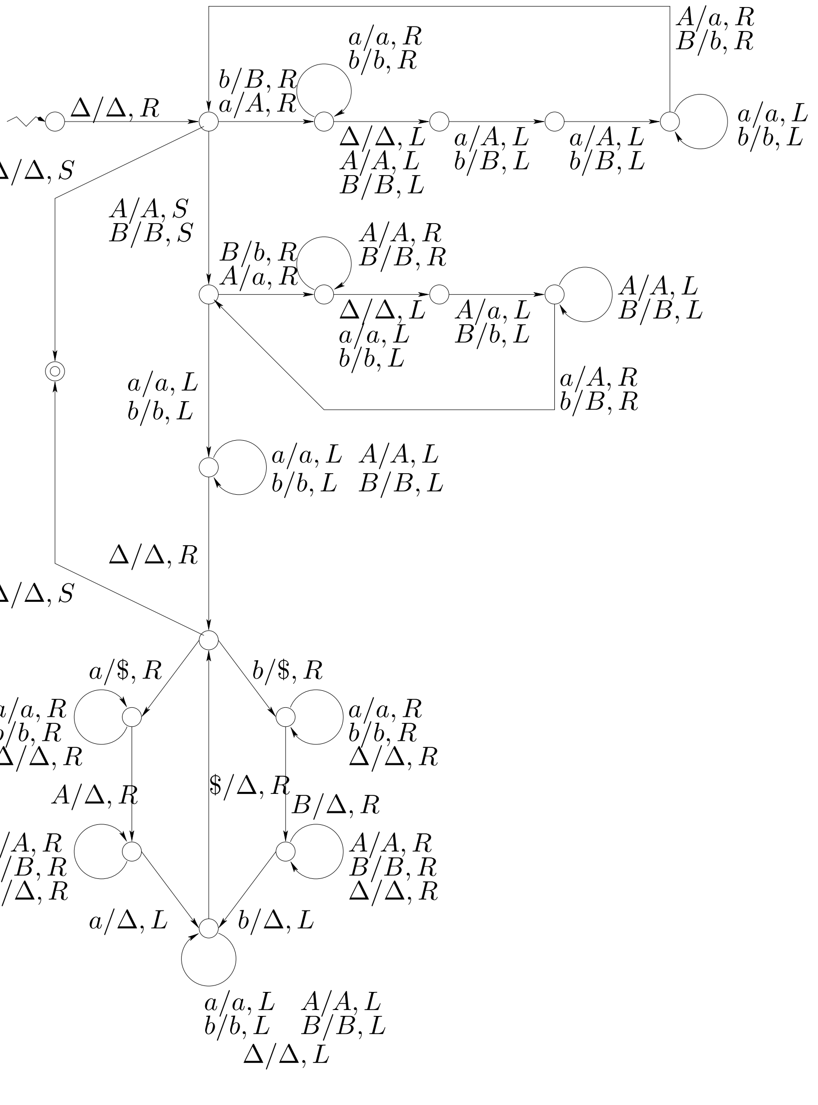
9.6
d We consider the language L consisting of balanced strings of parentheses, to be precise

$$L = \{w \in \{(,)\}^* \mid n_((w) = n_)(w) \text{ and } n_((u) \geq n_)(u) \text{ for every prefix } u \text{ of } w\}$$

A Turing machine accepting L could work as follows:
— mark cell 0 to avoid falling off the tape in a later stage: Δ is replaced by \$; next we go from left to right to the end of the input string (which is accepted immediately if it is empty, that is, cell 1 contains a Δ); the first (blank) cell right from the input is marked as \$ and the head moves back to the beginning; — after this bookkeeping the real work can begin:
move to the right to the first occurrence of a), erase it and move to the left and erase the first (thus encountered; repeat this procedure until
— either no right parenthesis is found anymore; thus the head hits \$ when walking to the right after which we go back to the left and reject if a (is seen, otherwise only Δ's are are seen until the \$ in cell 0 and we accept; — or when moving to the left no (is found anymore; we thus hit \$ in cell 0 and reject.
Draw a TM transition diagram for the algorithm just sketched.
How would you solve the problem if more types of parenthesis are used?

f In order to construct a Turing machine which will accept the language $\{www \mid w \in \{a, b\}^*\}$ we distinguish the following subtasks:
— determining whether the length of a given input word is a multiple of three (if not, it must be rejected);
— dividing the input accordingly in three consecutive subwords of the same length;
— determining whether these subwords are the same.

The exact implementation of the first two steps depends on the strategy which will be followed in the third step. An idea would be to adapt the approach of Example 9.3, Figure 9.5, to determine whether a word is of the form ss. Thus comparing for example the first and the second subword and if they are found to be equal then compare (one of) them with the third subword.
Here we propose to compare the three subwords in one stroke
and because of space considerations we first give the transition diagram:



— if the input is empty, we are done (accept);

— else, we first try to divide the word in one third vs two thirds by marking, for every symbol in the first part of the word (from left to right), two symbols in the last part (from right to left). As a marking we use A for a and B for b. Next we split the last part (consisting of capital letters) in half in such a way that the last (third) subword becomes unmarked again;
— now we are ready for the last phase with a word of the form wxy with $w, y \in \{a, b\}^+$ and $x \in \{A, B\}^+$ and $|w| = |x| = |y|$:
we move back to the beginning of the tape and begin the comparison: remember and mark (as \$) the first (non-blank) symbol, go to the right and compare it with the first capital symbol; if they agree erase this capital (replace it by Δ) and move on to the right to the first (small) symbol; again compare and if it is the same as the original symbol erase it and move back to the left of the input to the \$; delete this and move right;
if the symbol then read is a Δ we are done and can accept; else repeat the above procedure (if a comparison is not successful, the machine crashes/stops in h_r).

9.7 The TM in Figure 9.22 when given an input string 1^i repeatedly divides i by 2 until exactly one 1 remains after which it accepts. Hence the input is accepted if and only if i is a power of 2. Thus the TM accepts the language $\{1^{2^n} \mid n \geq 0\} = \{1, 11, 1111, 11111111, 1^{16}, \ldots\}$.

9.11 Let T be a TM accepting a language L. We modify T in such a way that the resulting TM T' also accepts L and never stops unsuccessfully (the reject state h_r is not needed, neither explicitly nor implicitly).
Recall that T stops unsuccessfully if and only if either it scans a symbol in a state for which combination it does not have an instruction or the head falls off the left side of the tape.

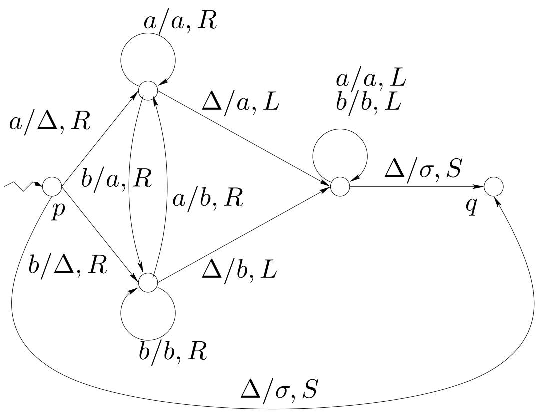
First we consider the problem of the head falling off the tape. We use symbols with a subscript L as an indication of their occurrence in the leftmost cell. We thus have a new symbol Δ_L and new tape symbols a_L for every $a \in \Gamma$ where Γ is the tape alphabet of T. We let T' begin its work on any input x by first changing the Δ in the leftmost cell into Δ_L. Thus from here (configuration $(q_0, \underline{\Delta}_L x)$) it behaves as T unless it sees a subscript L. For these cases, instructions $\delta'(p, a_L) = (q, b_L, d)$ are provided whenever $\delta(p, a) = (q, b, d)$ and d is either R or S (the head moves to the right or stays); for instructions $\delta(p, a) = (q, b, L)$ (when the head would move to the left and fall off), we do nothing. Note that T' (constructed sofar would crash by lack of instructions rather than falling off the tape. The set of accepted words has not been changed.

Finally, we add a new state `sink` together with instructions $\delta'(\texttt{sink}, a) = (\texttt{sink}, a, S)$ for all symbols $a \in \Gamma \cup \Gamma_L \cup \{\Delta\}$. Thus once the TM is in `sink` it will remain (loop) there forever scanning the same cell with the same symbol. For every combination (p, a) of a state p and a symbol a for which there is no instruction (thus in particular for $p = h_r$ which is now considered an ordinary state), we set $\delta'(p, a) = (\texttt{sink}, a, S)$. Hence whenever T would crash because of lack of an instruction (or fall off the tape), the modification now takes care that it moves to `sink` (and only then). Note that adding these instructions does not make the TM non-deterministic nor affects the set of words accepted.

9.12 As mentioned in the text (page 336), during (or after) a computation of a TM there is no (general, effective) procedure to determine the position of the rightmost cell containing a non-blank symbol. Simply walking to the right does not give insight in whether the last non-blank has been seen or whether there might still be another one. As we discuss now also more subtle methods will not work.
a Assume we have a module TM T_0 which when started on a tape will move the head to the rightmost position on the tape containg a non-blank (if the tape is completely blank it moves the head to cell 0) and then stop.
First consider a TM T_1 which halts (for some input) in the accepting configuration $(h_a, \underline{\Delta}1)$. Then T_0 begins its work and after some finite time it halts with its head on the 1 in cell 1 with the tape otherwise empty. We now make a TM T_2 which works as follows. It erases its input, moves back to the beginning of the tape and writes 1 in cell 1, then it invokes T_0 but with the following modification: it marks cell 2 with the endmarker # and whenever # is read it is treated as Δ and the marker is shifted one cell to the right. In this way there will always be exactly one # on the tape directly after the rightmost cell ever visited by T_0. Once (the thus) modified version of T_0 has finished its work we stop in the configuration $(h_a, \underline{\Delta}1\Delta^n\#)$ with cell $n+1$ the rightmost cell ever visited by T_0.
Now we ask T_0 to find the rightmost non-blank cell left by T_2. Since it is deterministic it will proceed as for T_1 beginning from $(h_a, \underline{\Delta}1)$ and will never visit the cell with #. Thus T_0 does not work correctly for T_2.

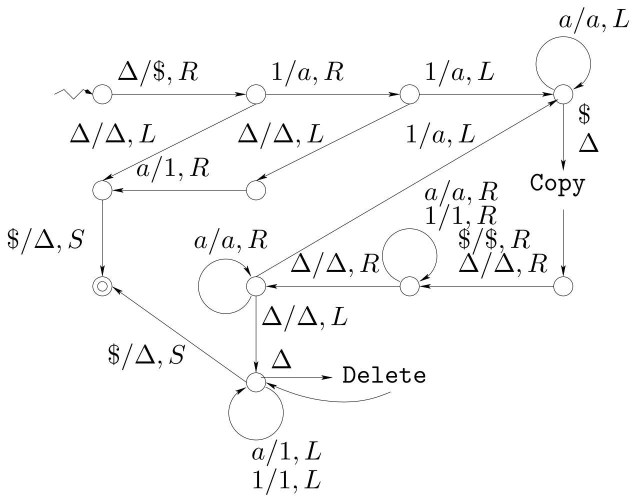
b If a TM is modified in such a way that it always marks (as described above) the rightmost edge of the portion of the tape it has visited, then once it has finished its computation, a brute force method of simply walking to the right would work: start from the leftmost cell and walk to the right until that (unique) marker and then go back to the left and the rightmost cell left non-blank during that computation will be found.

9.13 We construct a (module) Turing machine `Insert`(σ) with input alphabet Σ which places the symbol $\sigma \in \Sigma \cup \{\Delta\}$ at the current position of the tape head and shifts the tape contents which follows one position to the right.
More precisely: $(p, y\underline{z}) \vdash^* (q, y\underline{\sigma}z)$ where p is the initial state of `Insert`(σ) and q its terminating state (h_a if you like); z does not contain Δ's.
The TM achieves this by writing a Δ at the current position if that is not blank (if it is, we know that $z = \Lambda$ and the TM can write σ rightaway and stop); the TM moves to the right, all the time replacing the next symbol by the previous one. When it sees Δ it writes the final symbol, moves back to the left until it sees the blank cell left at the original beginning of z where it writes σ and stops.
Below follows a detailed transition diagram for the case that the input alphabet of the Turing machine consists of a, b only (and so $\sigma \in \{a, b\} \cup \{\Delta\}$). Generalizing this to arbitrary alphabets should be easy.



9.15
c We construct a TM that computes the square of any given positive integer (in unary), thus we aim at $(q_0, \underline{\Delta}1^n) \vdash^* (h_a, \underline{\Delta}1^{n^2})$.
Basically given an input string of n 1's we have to produce another $n-1$ strings of n 1's. Therefore we will use the modules `Copy` and `Delete`, the latter to remove the redundant Δ's inbetween the copied strings.
In order to distinguish the first Δ in cell 0 from other occurrences of Δ we change it till the end of the computation in \$. This implies that we use a slightly extended version of the module `Copy` as described in Example 9.7 (Figure 9.12). The modified version used here treats \$ as a Δ, that is $(p, \underline{\Delta}x) \vdash^* (q, \underline{\Delta}x\Delta x)$ and also $(p, \underline{\$}x) \vdash^* (q, \underline{\$}x\Delta x)$ with p the initial state of

Copy and q its final state. Also this modified version of Copy does not visit any other cells than those used in the final configuration. (Give a transition diagram for this Copy.)
The module Delete is given in Example 9.8 (Figure 9.13); $(p, y\underline{a}z) \vdash^* (q, y\underline{z})$ where z doesn't contain blanks and is followed by a blank, a may be a blank, and p and q are the initial and final state. This Turing machine only visits the cells occupied by $az\Delta$, the last part of its input.
Our TM works as follows:
If the TM is given 0 as input, it can stop immediately (after changing \$ back into Δ), because $0^2 = 0$.
If the input is not empty, the first 1 is marked (as a) and the TM looks for a second 1. If there is none it changes the a back into 1 and the \$ into Δ and it stops, because $1^2 = 1$.
If the TM discovers a second 1, then it proceeds as follows:
the 1 is marked, the head moves back to \$ and Copy copies the current string. The head moves to the right to the beginning of the fresh copy and looks for an unmarked 1. If there is one, it is marked, the head moves back to the beginning of this block and it is copied. This procedure is repeated until there is no 1 left in the last copied block, after which the head moves to the left, on the way deleting the Δ's and changing the a's into 1's until \$ is reached. This is changed into Δ and the machine stops.
The transition diagram is given below.



d Modify (and take care of a proper final configuration) the TM in Figure 9.22 (see Exercise 9.7).

9.16 Note that in Figure 9.23, the big arrow labelled $\Delta/\Delta, R$ has two arrowheads; it should however point only to the left.
Compare the TM shown with the solution to Exercise 9.15(i)!

9.17 We have two Turing machines T_1 and T_2, computing the functions f_1 and f_2, respectively. We assume that both functions have a single argument. For questions **a** and **b** we first describe a composite Turingmachine to compute $f_1(x)$ and $f_2(x)$ using T_1 and T_2 given some input x.
Copy the input x and insert a marker \$: $(q_0, \underline{\Delta}x) \vdash^* (q_{01}, \Delta x\$\underline{\Delta}x)$;
Compute f_1 using T_1: $(q_{01}, \Delta x\$\underline{\Delta}x) \vdash^* (h_1, \Delta x\$\underline{\Delta}f_1(x))$;
Interchange x and $f_1(x)$: $(h, \Delta x\$\underline{\Delta}f_1(x)) \vdash^* (q_{02}, \Delta f_1(x)\$\underline{\Delta}x)$;
Compute f_2 using T_2: $(q_{02}, \Delta f_1(x)\$\underline{\Delta}x) \vdash^* (h_2, \Delta f_1(x)\$\underline{\Delta}f_2(x))$.
Delete the marker, go to the beginning of the tape and stop:
$(h_2, \Delta f_1(x)\$\underline{\Delta}f_2(x)) \vdash^* (h_a, \underline{\Delta}f_1(x)\Delta f_2(x))$.
Thus, apart from T_1 and T_2 we need TM modules to copy (see Example 9.7), insert (Exercise 9.13), delete (Example 9.8), and to swap two strings (design your own TM module for this task).

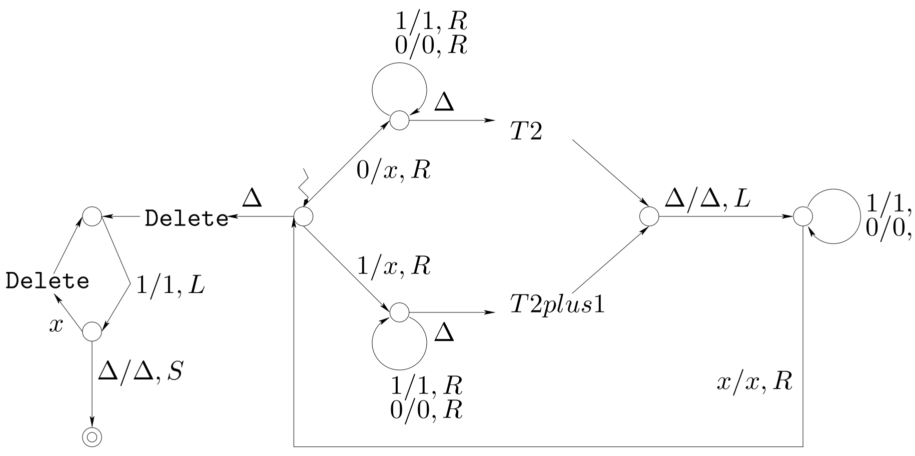
a Now we can add $f_1(x)$ and $f_2(x)$, assuming that we have a module to add two natural numbers (represented in unary, binary, ...):
$(q_+, \underline{\Delta}f_1(x)\Delta f_2(x)) \vdash^* (h_a, \underline{\Delta}f_1(x) + f_2(x))$.

b Compare $f_1(x)$ and $f_2(x)$:
$(q_{min}, \underline{\Delta}f_1(x)\Delta f_2(x)) \vdash^* (h_a, \underline{\Delta}min(f_1(x), f_2(x)))$.
This TM should still be defined!

c The function $f_1 \circ f_2$ is defined by $f_1 \circ f_2(x) = f_1(f_2(x))$ for all x in the domain of f_2. It is undefined if $f_2(x)$ is not in the domain of f_1. We assume that the tape alphabet of T_2 is contained in the input alphabet of T_1.
To compute $f_1 \circ f_2$ given some input x, first compute $f_2(x)$ using T_2: $(q_{02}, \underline{\Delta}x) \vdash^* (h_2, \underline{\Delta}f_2(x))$ if $f_2(x)$ is defined (otherwise, the TM T_2 crashes or doesn't stop at all);
then apply T_1 to $f_2(x)$ with $h_2 = q_{01}$: $(q_{01}, \underline{\Delta}f_2(x)) \vdash^* (h_a, \underline{\Delta}f_1(f_2(x)))$ if $f_1(f_2(x))$ is defined.

9.18 Let us assume that we are given two Turing machines $T2$ and $T2plus1$:
$T2$ computes the function $f_2(1^k) = 1^{2k}$, for all $k \geq 0$, and
$T2plus1$ computes the function $f_{2+1}(1^k) = 1^{2k+1}$, for all $k \geq 0$. We can now construct a TM which when given an arbitrary binary string (which may start with leading 0's) computes in unary the number it represents:
Going from left to right through the input string we compute after the blank following the input (starting from $0 = \Delta$) in unary 2 times the current value

for each 0 and 2 times the current value +1 for each 1 we encounter. Once the whole input has been processed (and marked) in this way, we delete the Δ between marked input and result and then moving to the left we delete all x's and stop with the head on the blank in cell 0.
See the transition diagram.



9.19 We assume that we already have a Turing machine T_{+1} which when given any natural number n in binary representation $\texttt{bin}(n)$ (which is either 0 or starts with a 1) computes the binary representation of $n+1$:
$(q_0, \underline{\Delta}0) \vdash^* (h_a, \underline{\Delta}1)$ and $(q_0, \underline{\Delta}\texttt{bin}(n)) \vdash^* (h_a, \underline{\Delta}\texttt{bin}(n+1))$ for all $n \geq 1$.
A Turingmachine that converts unary into binary could now work as follows:
Input 1^k;
if $k = 0$ (empty tape), write 0 (in cell 1), move to cell 0 and stop;
otherwise (there is at least one 1), move to the end of the input string, change the last 1 into a marker x, move two cells to the right, write a 1 and go back to the right end of the remaining input;
repeat, until no input 1 remains, for each rightmost 1 in the remaining input: replace 1 by x, move to the right until the first Δ, apply T_{+1}, go back to the right end of the remaining input;
if all input has been thus dealt with, change the tape contents from $\Delta x^k \Delta\texttt{bin}(k)$ into $\Delta\texttt{bin}(k)$ and stop with the head on cell 0.
Draw the transition diagram.

9.21 In Example 9.3, Figure 9.5, a TM has been given for the language $L = \{ss \mid s \in \{a,b\}^*\}$. We now sketch a TM that accepts L, this time without using any additional tape symbols ($\Gamma = \Sigma$):

Given is an input word $x \in \{a,b\}^*$.
If $x = \Lambda$ (the tape is empty) we can accept immediately. Thus from here we assume that $x = c_1 \ldots c_k$ with $c_i \in \{a,b\}$ and $k \geq 1$.
First it is checked that x is of even length:
insert an extra blank in cell 1: $(p, \underline{\Delta}x) \vdash^* (q, \Delta\underline{\Delta}x)$;
move c_1 one cell to the left and c_k one cell to the right; repeat this procedure for $c_2 \ldots c_{k-1}$ until either it turns out that k is odd (no symbol found to move to the right) in which case x is rejected; or we end with tape contents $\Delta c_1 \ldots c_{k/2} \Delta\Delta c_{(k/2)+1} \ldots c_k$.
Next compare the two halves of x symbol by symbol more or less as before, but rather than using uppercase letters as in Example 9.3 we replace symbols that have been dealt with by Δ. Before each pass we have to check however whether or not the symbols to be compared are the last symbols. This to avoid that, in the next pass, the head falls off when we move to the left in search for a nonblank symbol.

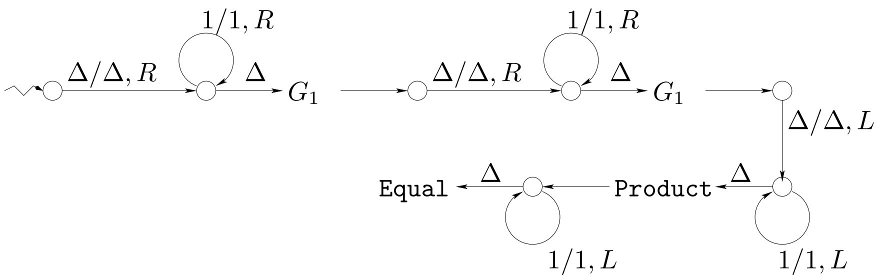
9.29 We are given the transition table of an NTM (draw the transition diagram if you like). Assume that it begins a computation with a blank tape. Then first $(q_0, \underline{\Delta}) \vdash (q_1, \Delta\underline{\Delta})$ and next there is a choice:
to write 0 or 1, move to the right, and continue in state q_1:
$(q_1, \Delta 0 \underline{\Delta})$ or $(q_1, \Delta 1 \underline{\Delta})$;
or to leave Δ as it is, move to the left, and change to state q_2.
Thus, unless the NTM moves forever to the right in state q_1 while writing 0's and 1's, it reaches the state q_2 after having carried out some finite number n of instructions and having written a word $w \in \{0,1\}^n$, and then moves to the left (deterministically):
$(q_1, \Delta\underline{\Delta}) \vdash^n (q_1, \Delta w \underline{\Delta}) \vdash^{n+1} (q_2, \underline{\Delta} w \Delta) \vdash (h_a, \underline{\Delta} w \Delta)$.
Thus when started with a blank tape, this NTM can have any binary string as output (or may never stop). The NTM crashes after its first step when started with a non-empty input x: $(q_0, \underline{\Delta}x) \vdash (q_1, \Delta\underline{x}) \not\vdash$.

9.30 G is the NTM from exercise 9.29 with $(q_0, \underline{\Delta}) \vdash^* (h_a, \underline{\Delta} w)$ where w can be any string over $\{0,1\}$.
`Copy` is the TM from Example 9.7 (Figure 9.12) with $(q_0, \underline{\Delta}x) \vdash^* (h_a, \underline{\Delta}x\Delta x)$ for all words $x \in \{0,1\}^*$.
`Equal` is a TM such that, for all $x, y \in \{0,1\}^*$, $(q_0, \underline{\Delta}x\Delta y) \vdash^* (h_a, v\underline{c}w)$ for some v, c, w if and only if $x = y$.
`Delete` is the TM from Example 9.8 (Figure 9.13) with $(q_0, y\underline{a}z) \vdash^* (h_a, y\underline{z})$ where z doesn't contain blanks and a may be a blank.
The NTM in Figure 9.25 when given an input word $u \in \{0,1\}^*$, first goes to

the right end of the input, then applies the NTM G starting from the first blank right from the input. Thus (nondeterministically) the tape contents are changed into $\Delta u \Delta w \Delta \dots$ with $w \in \{0,1\}^*$ an arbitrary word; the head is on the Δ inbetween u and w.
Next w is copied and the Δ inbetween the copies is deleted.
The head moves to cell 0 and with `Equal` it is tested whether $u = ww$.
Consequently, the language accepted by this NTM is $\{ww \mid w \in \{0,1\}^*\}$.

9.31 Hints to construct an NTM for $\{1^n \mid n = k^2 \text{ for some } k \geq 0\}$:
Use the NTM G from exercise 9.29, the TM `Square` from exercise 9.15(c), and the TM `Equal` as in exercise 9.30.
As in the previous exercise, generate an arbitrary string from the language given, compare it with the input and accept if and only if they are equal.
Thus: walk to the right end of the input word $x \in \{1\}^*$, generate with G an arbitrary string w, apply `Square` which crashes when it encounters a 0.
If the NTM has not crashed, then $w = 1^k$ for some $k \geq 0$ and we now have on the tape $\Delta x \Delta 1^{k^2}$. Finally, x and 1^{k^2} are compared.

9.32 An NTM for $\{1^n \mid n \text{ is a composite integer } \geq 4\}$ has to check whether the length of a given input from $\{1\}^*$ is at least 4 and not prime.
We let this NTM first generate non-deterministically a string of the correct form, after which it compares this string with the input.



The following modules are used (if not yet available, construct them!):
an NTM G_1 with $(q_0, \underline{\Delta} \vdash^* (h_a, \underline{\Delta}1^k)$ with $k \geq 2$ and which doesn't accept any non-empty input;
a TM `Product` with $(q_0, \underline{\Delta}1^k\Delta 1^j \vdash^* (h_a, \underline{\Delta}1^{k \cdot j})$;
the TM `Equal` as in exercise 9.30.
The NTM walks to the right end of the input word $x \in \{1\}^*$, generates with G_1 an arbitrary string 1^k with $k \geq 2$; moves again to the right and then calls G_1 again to obtain a string 1^j with $j \geq 2$; it goes back to the left to

the Δ preceding 1^k and calls `Product`. We now have on the tape $\Delta x \Delta 1^{k \cdot j}$. Finally, x and $1^{k \cdot j}$ are compared.

9.33 Let $L \subseteq \{0,1\}^*$ be a language accepted by a Turing machine T.
a To accept the set of prefixes of L, an NTM T_{pref} could begin by moving past the input x, call the NTM G from exercise 9.29 and generate an arbitrary word from $\{0,1\}^*$ and concatenate this to x by deleting the Δ inbetween x and w. It then moves back to cell 0 and calls T which will lead to acceptance if and only if $xw \in L$, thus if and only if x is a prefix of a word from L.
b As in **a**, but now the word w generated by G has to be moved to the left of x: we need a module $(q_0, \Delta x \underline{\Delta} w) \vdash^* (h_a, \underline{\Delta} wx)$. Then call T.
c First as in **a**, concatenate x with an arbitrary w leading to xw; then, as in **b**, concatenate an arbitrary v to xw which yields vxw. Then call T.

9.38 We are asked to describe a TM that enumerates the palindromes over $\{0,1\}$ in canonical order: $x \prec y$ iff $|x| < |y|$ or $|x| = |y|$ and x comes alphabetically before y.
We will use the fact that a palindrome v preceeds a palindrome w if and only if the first half of v (including its middle symbol if it is of odd length) preceeds the first half of w.
First we describe a module TM which when given a palindrome as input produces the next (in the canonical order) palindrome as output:
$(q_0, \underline{\Delta} x_n) \vdash^* (h_a, \Delta x_n \underline{\Delta} x_{n+1})$.
— use the module Copy to copy the input: $(q_0, \underline{\Delta} x_n) \vdash^* (q_1, \underline{\Delta} x_n \Delta x_n)$
— examine the copy:
if x_n contains no 0's, then change the copy to $0^{|x_n|+1}$, which is a palindrome; otherwise find the middle of the copy and change from there — going from right to left through the string — all 1's into 0's, until a 0 is found which is changed into 1; then modify the second half accordingly (palindrome); thus the copy of $x_n = y01^j0y^r$ has been changed into $y10^j1y^r$ (check that this is indeed x_{n+1}!);
— go back to the Δ just before the thus modified copy and stop.
Clearly we can now make a TM enumerating all palindromes in canonical order by first taking care that q_0 only occurs in the first step of a computation as just described and then changing h_a into q_0. Hence when started on the empty palindrome, the TM will never stop and only be busy creating on the tape a list of all palindromes in canonical order.

Note that we could also have given a module TM which replaces the input x_n itself, rather than a copy, by x_{n+1}, which then would have led to a TM which shows all palindromes one after the other (in time), rather than behind one another.

---

versie 06 januari 2009