Dino Mandrioli

Luigi Lavazza

Angelo Morzenti

Pierluigi San Pietro

Paola Spoletini

Exercises of Theoretical Computer Science

Third edition revised, corrected and extended

Preface

This workbook is the result of the efforts to correct and improve the first two editions, respectively of the 1994 and the 2001. Many oversights and inaccuracies have been eliminated, several solutions have been clarified and discussed more extensively. Moreover, it has been added a wide choice of exam texts of Theoretical Computer Science of the Politecnico di Milano relative to the period 1995-2004.

The workbook has been conceived and realized to cover the needs of the Theoretical Computer Science course of the Poltecnico di Milano and, partially, of the second level course “Analisi e progetto di sistemi critici”. However, the treated topics are the foundation of any academic course that aims to illustrate the theoretical foundations of computing, so the book can be effectively used outside the Polytechnic.

The authors have referred, for the terminology and the basic definitions, at the book *Informatica Teorica* of Carlo Ghezzi and Dino Mandrioli, edited by UTET. The book, however, should be accessible to anyone already familiar with the topics addressed therein.

The exercises marked with one star are more difficult than normal. Those marked with two stars are much more difficult than average.

Index

1. Models and their properties

1.1 Finite state automata

Exercise 1.1.1

Consider a three-position electrical switch, as shown in **Figure 1-1**. The switch consists of a knob that can rotate in one of the three positions A, B, C, after a shift in a clockwise or counterclockwise. Model the operation of the switch with a finite automaton. Consider then a switch that is initially in position A, which sequences of moves clockwise and counterclockwise bring in the position C?

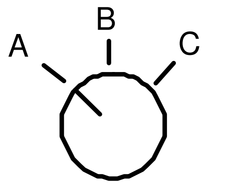


Figure 1-2 Three-position switch.

Solution

An FA models the possible configurations of a system with *states*, and events or external actions (inputs) that can cause a configuration change with *transitions*. An automaton that solves the proposed exercise, shown in **Figure 1-3**, has a set of states Q = (A, B, C) to represent the position and a transition function δ that models changes in position of the switch following the application of an appropriate motion to the knob.

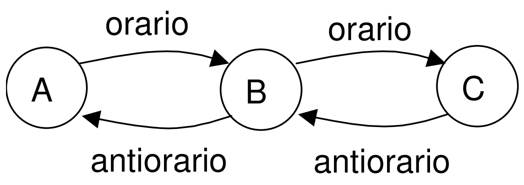

$$\begin{tikzcd} A \arrow[r, bend left, "orario"] & B \arrow[l, bend left, "antiorario"] \arrow[r, bend left, "orario"] & C \arrow[l, bend left, "antiorario"] \end{tikzcd}$$

Figure 1-4 A first finite state model for the switch ("Orario" = clockwise, "Antiorario" = counterclockwise)

Note that there are edges outgoing from the state A in the case of applying a momentum counterclockwise and state C for a momentum. In fact you do not model the possibility of rotation beyond the positions A and C. This aspect was not considered in the simple informal description given in the exercise, which is based on common knowledge of similar systems. The need to construct a formal model forces us to make explicit the assumptions "hidden" on the system and eventually to choose between various options.

In fact there are several solutions. For example, the switch may be circular in the sense that a clockwise rotation applied to C causes the shift in position A and a counterclockwise applied to A cause the transition in C. Another possibility is that the attempt to turn the switch over positions A or C results in the breaking of the switch itself. An automaton to model this situation has a failure state G, which for simplicity has no expected recovery. Another solution is that the switch cannot rotate beyond the positions A and C, and any attempt to further rotate is ignored (it remains in states A and C). The presence of self loops in states A and C models a possible choice in the construction of the switch.

See **Figure 1-5Error! Reference source not found.**.

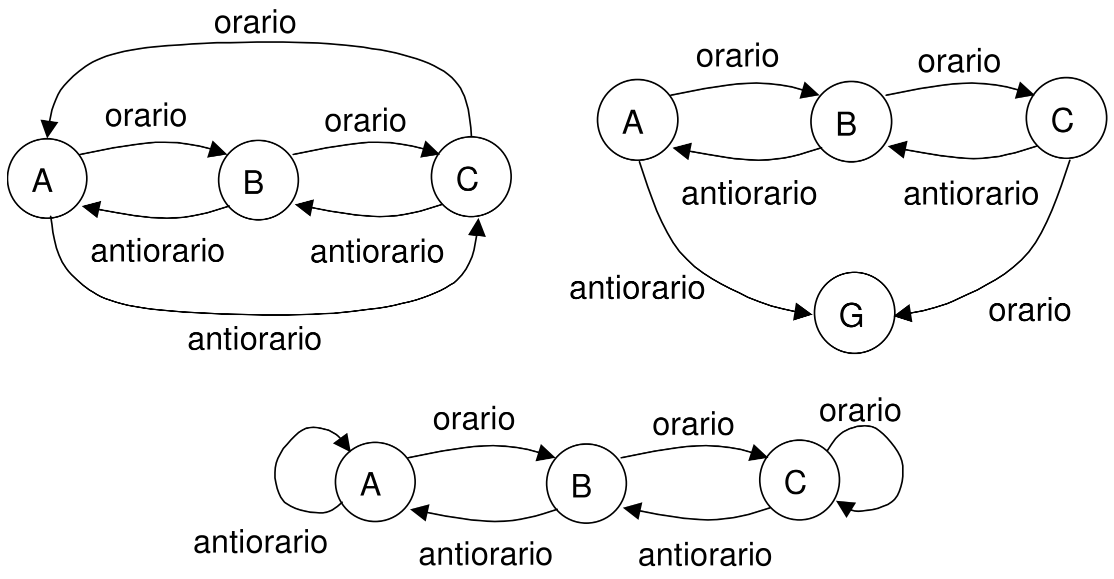


Figure 1-6 Other models for the switch. ("Orario" = clockwise, "Antiorario" = counterclockwise)

Formal models allow to study properties of the system modeled. Assuming that you choose a model in **Figure 1-7**, *a* denote a counterclockwise movement, and with *o* a clockwise movement. The sequences of moves bring to C from A represent the language accepted by the automaton of **Figure 1-8**.

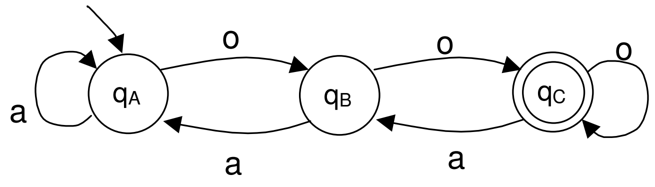


Figure 1-9 An automaton acceptor of the language sequences of moves that lead from A to C

The automaton allows one for example, given a sequence of moves, to easily determine if the switch is in position C at the end of the sequence. You can also give a more intuitive definition for L: it is the language of s sequences in which there is at least a suffix of s that contains at least two or more of a's.

From the automaton one can also obtain directly a regular expression for L:

$$a^*\ o\ (a^+\ o)^*\ o\ ((a\ (a^+\ o)^*\ o)^*\ o^*)^*$$

The expression can be simplified using the identity $(x^*\ y^*)^* = (x \mid y)^*$ valid for any regular expression x and y, obtaining the expression:

$$a^*\ o\ (a^+\ o)^*\ o\ ((a\ (a^+\ o)^*\ o) \mid o)^*$$

Exercise 1.1.2

Build some DFA automaton that recognize the following languages:

$L_0 = \{x \mid x\in \{0,1\}^*$ and any 0 in x is followed by at least a 1$\}$. Strings example: 010111, 1111, 01110111011.

$L_1 = \{x \mid x\in \{0,1\}^*$ and x ends with 00$\}$;

$L_2 = \{x \mid x\in \{0,1\}^*$ and x contains exactly 3 zeros$\}$;

$L_3 = \{x \mid x\in \{0,1\}^*$ and x starts with 1$\}$;

$L_4 = \{x \mid x\in \{0,1\}^*$ and x does not begin with 1$\}$;

Solution

The acceptor automaton for L_0 is shown in **Figure 1-5**. One can observe that the transition function used is partial, because $\delta(q1, 0)$ it is not defined. As a consequence our language will not accept strings with two or more consecutive zeros. When the second zero is found, the automaton stops itself without accepting the string, because it can not execute any move.

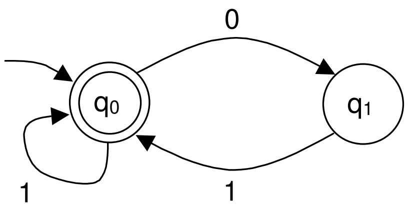


Figure 1-5 Automaton for the language L_0.

Automata for L_1, L_2 L_3 are shown respectively in **Figure 1-6**, **Figure 1-7** and **Figure 1-8**.

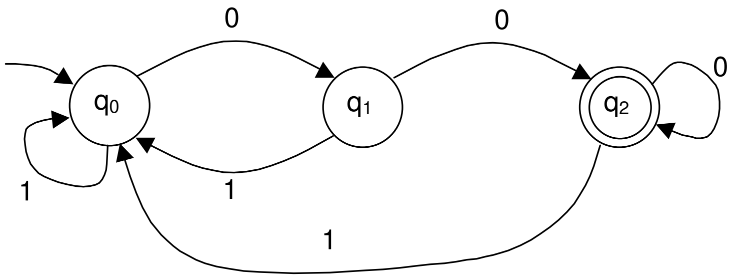


Figure 1-6 Automaton A1 for the language L1.

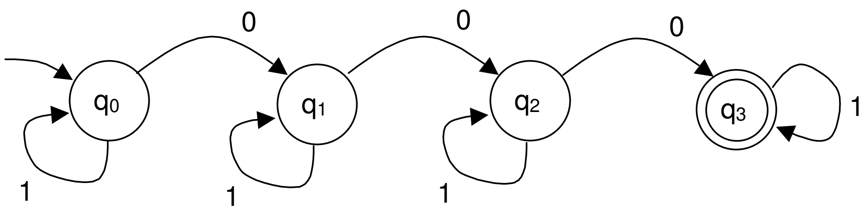


Figure 1-7 Automaton A2 for L2.

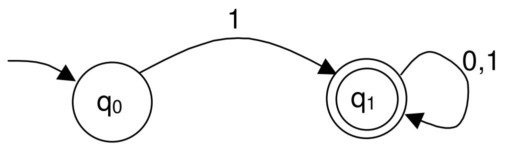


Figure 1-8 Automaton A3 for L3.

We can build the automaton A_4 that recognize L_4, completion of L_3, starting at A_3. First of all, we transform the transition function of the automaton A_3 form partial to total, because in the q_0 state there is not any edge corresponding to the input 0, adding to A_3 another state q_2, then we switch final and non final states. The result is shown in **Figure 1-9**.

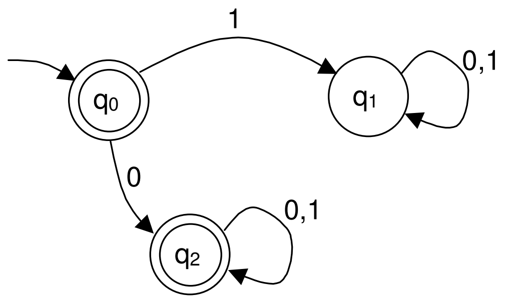


Figure 1-9 Automaton A4 per L4.

Exercise 1.1.3

Define the finite state automaton that recognize the following language L = $\{a^{2n}b^{3m}c \mid n \geq 1, m \geq 0\}$.

Solution

See **Figure 1-10**.

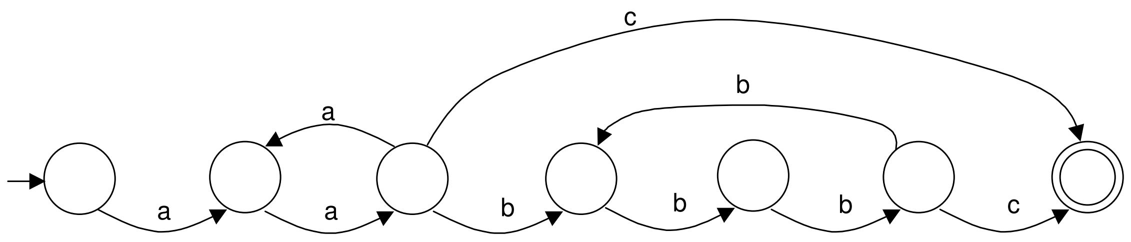


Figure 1-10 Finite state automaton for L= $\{a^{2n}b^{3m}c \mid n \geq 1, m \geq 0\}$.

Exercise 1.1.4

Define the finite state automaton that recognize the language L^*, where L is the language $\{a^{2n}b^{3m}c \mid n \geq 1, m \geq 0\}$.

Solution

See **Figure 1-11**.

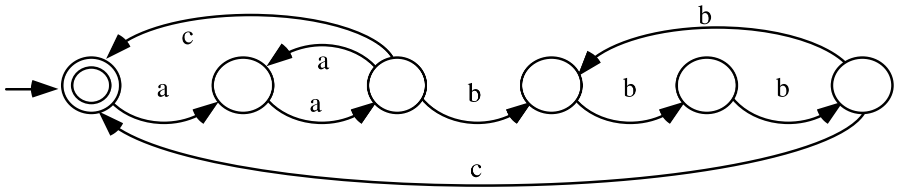


Figure 1-11 The finite state automaton of L*, where L is the language $\{a^{2n}b^{3m}c \mid n\geq 1, m\geq 0\}$.

Exercise 1.1.5

The **Figure 1-12** shows a flipper where the ball enters one of two top openings (A e B) and it exits from one of two bottom exits (C o D). the ball is diverted to right or left by the levers depending of their orientation when it passes throught the crossings; the levers change position (rotated of 90 degrees) when the ball comes through. For example, if the flipper is the configuration like **Figure 1-12** (a) the ball enters by B and it comes out from C and it leaves the levers in the position shown in **Figure 1-12** (b). Create the described device using the best automata.

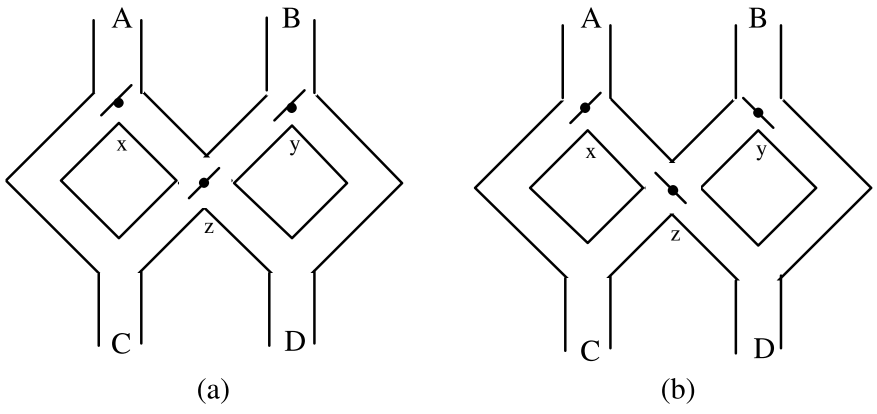


Figure 1-12 The flipper.

Solution

The most appropriate device is a finite transducer whose state is given by the position of the levers, whose input is given by the entrance opening of the ball and whose output by the exit opening. We indicate with the name x,y,z the state components (binary ones, in which 0 means right inclination and 1 means left inclination) which represent, in order, the position of the top left lever, of the top right lever and of the bottom one. We then assign to each of the eight feasible configurations a decimal number *s* from 0 to 7. In this way we obtain the transducer in **Table 1-1**, whose last two columns contain the pairs (next state, exit) which correspond to the current state, given by the row, and to the input, given by the column. For example, when the transducer is in state 3 (levers x,y,z, towards right, left and left), with input B gives output D and goes into state 1 (levers towards right, right and left).

x y z	s	A	B
0 0 0	0	(4, C)	(3, C)
0 0 1	1	(5, C)	(2, D)
0 1 0	2	(6, C)	(0, D)
0 1 1	3	(7, C)	(1, D)
1 0 0	4	(1, C)	(7, C)
1 0 1	5	(0, D)	(6, D)
1 1 0	6	(3, C)	(4, D)
1 1 1	7	(2, D)	(5, D)

Table 1-1 *The transducer which represents the pinball in* ***Figure 1-****12.*

Exercise 1.1.6

1. Please define a DFA which recognizes the following language:
   L = {x | x ∈ $\{0,1\}^*$ and the number whose binary representation is given by x can be divided by 3 without remainder}.
2. Please build a finite state transducer which computes the following function τ, for each x ∈ $\{0,1\}^*$: $\tau(x\$) = x \bmod 3$ in decimal representation.
3. Is it possible to compute the transduction $\sigma(x) = \tau(x\$)$ (without using the termination character $ in order to mark the end of the input) by means of a FT?

Solution to point 1.

First of all, we observe that adding a 0 bit in the least significant position to the binary representation of a number *i* we obtain the binary representation of the number *2i*, while if we add a 1 bit we obtain the binary representation of *2i+1*.

Given p = i mod 3, then, (2i) mod 3 = 2(i mod 3) mod 3 = 2p mod 3, and (2i+1) mod 3 = ((2i mod3) +1)mod 3 = (2p+1) mod 3.

The remainder *p* can be only one between 0, 1, 2, to which, respectively, correspond these values of 2p mod 3: 0, 2, 1, while 2p+1 mod 3 is 1, 0, 2.

It is enough to build a 3-states automata, each state corresponding to one of *p*'s values. The final state corresponds to null remainder. The automata is in state q_j iff the portion of the input that has been read until that moment has remainder *j* (meaning in q_0 p = 0, in q_1 p = 1, in q_2 p = 2).

The automaton is defined in **Figure 1-13**:

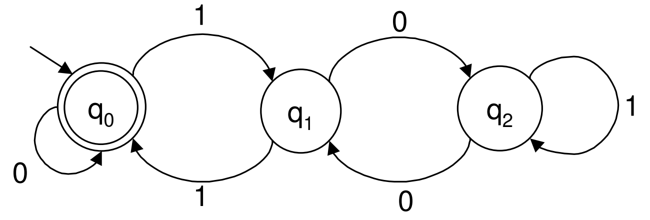


Figure 1-12 The DFA which recognizes L.

Solution to point 2.

A finite state transducer that computes the function can be easily definited from the previous automaton. We only need to add a state q3, the only final state, that can be reached from the other states with an edge marked with a \$ as input and the correct value of the remainder as output. All the others arcs have ε as output value. See **Figure 1-14**.

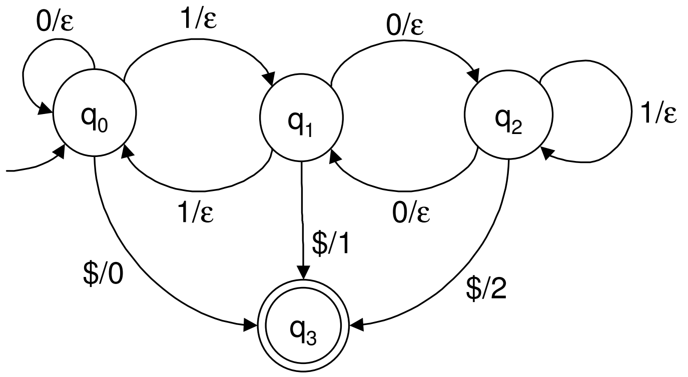


Figure 1-13 A finite state transuducer for τ.

Solution to point 3.

It is impossible to build a deterministic transducer for σ. An intuitive justification comes from the impossibility of doing a move that writes the correct value only when the whole string has been read. In the τ case , it is possible doing the move because there is the terminator % that allows to assign the end of the string. Without a terminator we need to build an automaton that for each bit *b* read as input, establishes whether *b* is the last b (we need to emit a correct value) or there are other bits to be read (in this case we need to continue). There is no finite state a transducer that recognizes σ because it is impossibile to choose between these two situations.

Exercise 1.1.7

Show that the automaton described in **Figure 1-14** accepts the language $L = \{x \in \{0,1\}^* \mid x = 1^{2k+1}$ or $x = y01^{2k+1}$, with $k \geq 0$, $y \in \{0,1\}^*\}$. Is the automaton A minimum?

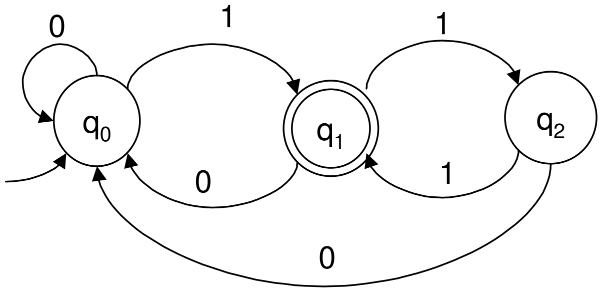


Figure 1-15 The automaton A.

Solution

The demonstration that L = L (A) can be carried out by induction on the length of the string. The inductive hypothesis corresponding to L = L (A) is that, given an n>1, for all $x \in \{0,1\}^*$, with 1<|x|<n, $\delta(q_0, x) = q_1$ if, and only if, x ends with an odd number of 1. This hypothesis alone is not sufficient to prove the thesis. An attempt to build the proof is in fact consider a generic string of length n of as a symbol preceded by a string of length *n-1*, to which to apply the induction hypothesis. The hypothesis can be applied only to strings that lead to the state q1, and in general can lead to a string any of the three states q0, q1 and q2. Then need a different inductive hypothesis for each state of the automaton.

Given a n>1, for every $x \in \{0,1\}^*$, with 0 ≤|x|<n:

1. $\delta(q_0, x) = q_0$ if, and only if, x does not end with 1, so ends with a zero number of 1;

2. $\delta(q_0, x) = q_1$ if, and only if, x ends with an odd number of 1;

3. $\delta(q_0, x) = q_2$ if, and only if, x ends with an even number of 1.

The base case is obvious, because: $\delta(q_0, \varepsilon) = q_0$, $\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$, $\delta(q_0, 11) = q_2$.

The inductive step is shown separately for each case:

1. x does not end in 1 ⇔ x is y0 type ⇔ $\delta(q_0, x) = \delta(q_0, y0)$ = (by definition of δ) $\delta(\delta(q_0, y), 0)$ = (being $\delta(q_i, 0) = q_0$ for i=0,1,2) q_0.
2. x ends with an odd number of 1⇔ x is of the type v1, with v that does not end in a, or is the type w1 with w that ends with an even number of 1 ⇔ $\delta(q_0, x) = \delta(q_0, v1)$ = (by definition of δ) $\delta(\delta(q_0, v), 1)$ = (by induction hypothesis on v) $\delta(q_0, 1) = q_1$ or $\delta(q_0, x) = \delta(q_0, w1)$ = (by definiton of δ) $\delta(\delta(q_0, w), 1)$ = (by induction hypothesis on w) $\delta(q_2, 1) = q_1$.
3. x ends with an even number of 1 ⇔ x is of the type z1, with z ends with an odd number of 1. ⇔ $\delta(q_0, x) = \delta(q_0, z1)$ = (by definiton of δ) $\delta(\delta(q_0, z), 1)$ = (by induction hypothesis on z) $\delta(q_1, 1) = q_2$.

This concludes the proof.

The automaton is not minimal, since, as can be easily understood by following the above test, the states q0 and q2 together represent strings ending in an even number or no 1. The minimal automaton A' is shown in **Figure 1-16**. As the number of states of A 'is less than A, one could minimize A obtaining A', and then verify that L = L (A '). The proof would be easier because there are only two instead of three inductive assumptions.

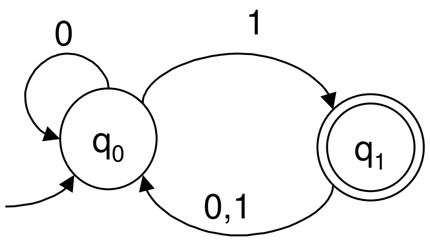


Figure 1-17 Minimum automaton A' for L.

Exercise 1.1.8

Build an FT computing transduction τ defined as:

$\tau((ab)^n ccc(ba)^m) = d^{2n}ef^{m \text{ div } 2}$, with $n\geq1$, $m\geq0$, and undefined for other input values.

Solution

Solution is described in **Figure 1-18**. This can be obtained considering that the automaton recognize the underlying language $L = \{(ab)^n ccc(ba)^m \mid n\geq1,\ m\geq1\}$, and meanwhile give appropriate output values to generate the desired translation. Note that in this case the automaton below is not the simplest automaton that recognizes L, because q7 and q8 are redundant for recognition (it is enough to put $\delta(q_6, a) = q_5$). They cannot be eliminated, however, from the transducer, because of the need to generate properly $f^{m \text{ div } 2}$: every two groups *ba* should generate one *f*.

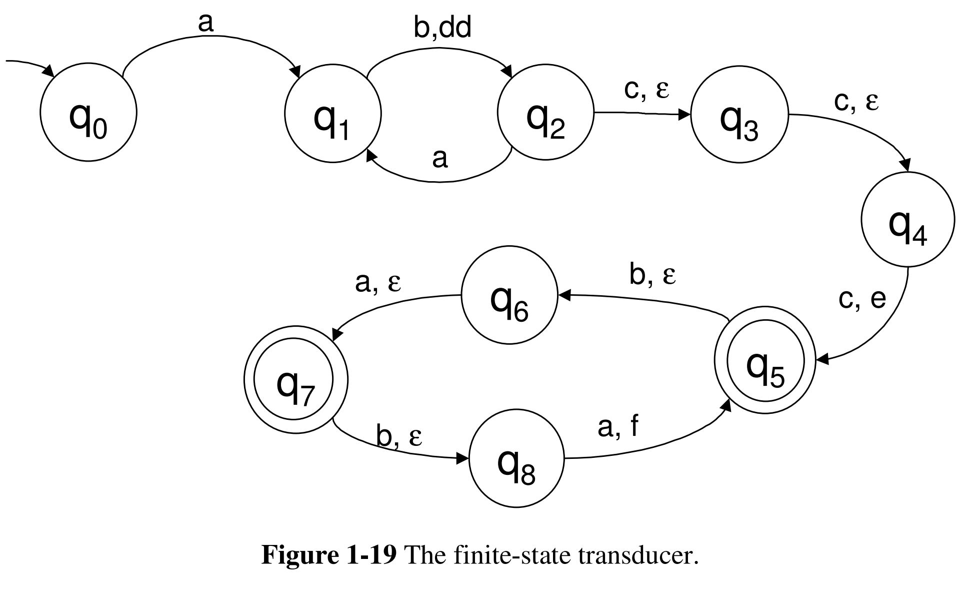


Figure 1-19 The finite-state transducer.

Exercise 1.1.9

A two-tapes finite deterministic automaton (2FA) has 2 tapes with head in input and it accepts pair of string. The set of states Q is partitioned into two subsets Q_1 e Q_2, and it moves one of its heads depending of the set to which the current state belongs to. For example, the

the automaton shown by the state diagram of **Figure 1-20** accepts every pair of strings (w_1, w_2) $\in$ $\{a, b\}^* \times \{a, b\}^*$ such that $|w_2| = 2 \cdot |w_1|$.

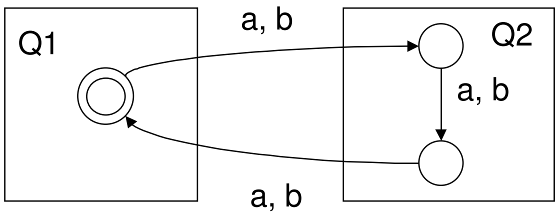


Figure 1-21 2FA that accepts $\{(w_1, w_2) \mid |w_2| = 2 \cdot |w_1|\}$. The labeled arcs "a,b" are a shothand notation for a pair of separated labeled arcs, respectively, "a" e "b". The automaton reads only a character by one of the tapes for each move.

1. Define by a state diagram the 2FA that accepts the following languages

i. each pair of strings (w_1, w_2) $\in$ $\{a, b\}^* \times \{a, b\}^*$ such that $|w_2| = |w_1|$ and $w_1(i)$=a if and only if $w_2(i)$=b and, viceversa, $w_1(i)$=b if and only if $w_2(i)$=a (for each string x, we assume that it is defined the function x(i) that gives the ith character of x: for example, if x=abbaba then x(3)=b e x(6)=a);

ii. (w_1, w_2) $\in$ $\{a, b\}^* \times \{a, b\}^*$ such that $|w_2| = 2 \cdot \#a(w_1) + 3 \cdot \#b(w_1)$, where #a and #b are functions that count the number of 'a' and 'b' of a string.

2. Formalize the following concepts

i. 2FA;

ii. a configuration of a 2FA;

iii. relation '$\vdash$' of transition between configurations of a 2FA;

iv. language accepted by a 2FA.

3. Study the relationship between the power of the model 2FA and the finite state automata (FA) and, optionally, the pushdown automata (PDA). We assume that a string x is esamined by a FA or an PDA is codified, if it is esamined by an 2FA, with the pair (x, ε), while a pair of strings (x, y) esamined by an 2FA is codified, if esamined by an FA or a PDA, by the string x·,·y (where '·' represents the concatenation operator between strings and the comma symbol ',' does not belong to the alphabet of the 2FA).

Solution to point 1.

See **Figure 1-22**.

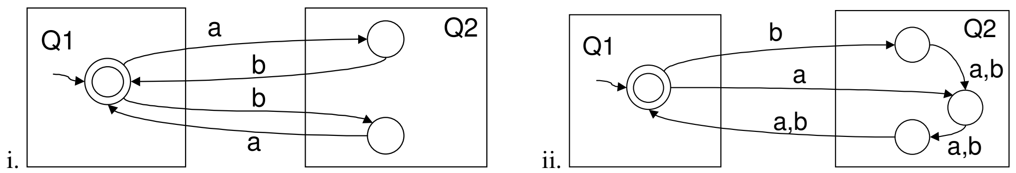


Figure 1-23 Solution to point 1 Exercise 1.1.9.

Solution to point 2.

i. A 2FA is a quintuple $<\Sigma, Q, \delta, q_o, F>$, where
   Σ is the input alphabet
   Q is the set of states, with $Q = Q1 \cup Q2$, and $Q1 \cap Q2 = \varnothing$;
   $\delta: Q \times \Sigma \rightarrow Q$ is the transition function, possibly partial;
   $q_o \in Q$ is the initial state;
   $F \subseteq Q$ is the set of final states;
ii. A configuration of 2FA is a string xqy, where $x,y \in \Sigma^*$ and $q \in Q$;
iii. $xqy \vdash \underline{x}q'y$ if and only if $q \in Q1$, $x=i\underline{x}$, and $\delta(q, i)=q'$;
   $xqy \vdash xq'\underline{y}$ if and only if $q \in Q2$, $y=i\underline{y}$, and $\delta(q, i)=q'$;
iv. Define $\vdash^*$ as reflexive and transitive closure of $\vdash$. The language L accepted by a 2FA is $L = \{ (x, y) \mid x,y \in \Sigma^*$ and $xq_oy \vdash^* q$ for any $q \in F\}$

Solution to point 3

The model 2FA is more powerful than FA: if x is accepted by a FA, (x, ε) is accepted by a 2FA with an empty Q2 part of the state graph and Q1 isomorphic to FA, while it is easy to demonstrate that the pair of string (x, x) is accepted by a 2FA but the string x·,·x is not accepted by any FA.

The model PDA is not more powerful than 2FA because the string x·,·x is not accepted by any PDA; neither the model 2FA is more powerful than PDA, because strings like xcx^R are accepted by a PDA but (xcx^R, ε) is not accepted by any 2FA. The last sentence can be easily justified considerino that any language $L = \{(x, \varepsilon) \mid x \in L'\}$ can be accepted by a 2FA if and only if L' is accepted by a FA. As matter of fact to accept strings like (x, ε) the automaton cannot perform any move that determines a transition in a state of Q2. Thus it has to perform only moves inside of Q1 and acts like a FA, that cannot accept the language $\{xcx^R\}$.

Exercise 1.1.10

a. Give a formal definition of a bidirectional automaton and its operation. A bidirectional automaton is a finite state machine where the reading head can also move to the left or remain still during a transition. The machine is used only in reading, just to recognize the languages.
b. Tell, justifying the answer, if the power of a bidirectional automaton decreases by eliminating the possibility of keeping the reading head still during a move.

Solution to point a.

A bidirectional automaton is a quintuple $<Q, I, \delta, q0, F>$, where Q, I, q0 and F are defined as for the traditional finite state automata. δ is instead a function $\delta: Q \times I \rightarrow Q \times \{D, H, S\}$ (where symbols D, H, S indicate the shift of the head: in order, Right, Halt, Left).

The functioning of a two-way automaton is defined through the notion of configuration and the relation of transition between two configurations.

A configuration c of the automaton is a pair $<q, w\uparrow y>$, $q \in Q$, $w, y \in I^*$, $\uparrow \notin I^*$

$c = <q, x> \vdash c' = <q', x'>$ if and only if it one of the following conditions is verified:

i. $x = w\uparrow iy$, $x' = wi\uparrow y$ and $\delta(q, i) = <q', D>$;
ii. $x = wi\uparrow y$, $x' = w\uparrow iy$ and $\delta(q, i) = <q', S>$;

iii. $x = w\uparrow iy$, $x' = w\uparrow iy$ and $\delta(q, i) = <q', H>$;

$c = <q, w\uparrow y>$ is an initial configuration of the automaton if and only if $q = q_o$ and $w = \varepsilon$.

$x \in I^*$ is accepted by the automaton if and only if $<q_0, \uparrow x> \vdash^* <q', x\uparrow>$ with $q' \in F$.

The above definition is modified in a standard way to deal with the case nondeterministic automata.

Solution to point b.

The power of a bidirectional automaton does not decrease by eliminating the possibility of keeping the head still during reading a move. In fact, we first consider only the case where the automaton is deterministic: given a generic q_1 and a generic i, only one of the following circumstances can be verified during the operation of a bidirectional automaton:

i. there is a finite sequence of definitions
$\delta(q_1,i) = <q_2, H>, \ldots \delta(q_{r-1},i) = <q_r, H>, \delta(q_r, i) = <q_s, M>$ with $q_j \neq q_h$ for each $j\neq h$; $M\neq H$.
In this case change δ with $\delta(q_j,i) = <q_s,M>$ for each j, $1 \leq j \leq r$.

ii. there is a sequence of definitions
$\delta(q_1,i) = <q_2, H>, \ldots \delta(q_{r-1},i) = <q_r, H>$, with $q_j = q_h$ for any $j \neq h$.
In this case change δ with $\delta(q_j,i) = <q_E,D>$ for each j, $1 \leq j \leq r$, where q_E is an error state where δ is undefined for all i.

Obviously the new automaton is equivalent to the previous one and it does not follow any stationary move (where the head remains still). Note that the equivalence remains whether the stationary sequences of moves of the automaton meet the original and final states or this circumstance does not occur. In fact, the automaton has to reach its end to accept a string and it cannot happen after a stationary move.

The case where the automaton is nondeterministic is treated in a similar way taking into account only the fact that cases i. and ii. are not mutually exclusive.

1.2 Pushdown automata

Exercise 1.2.1

Describe DPDPA that recognize the following languages:

$L_1 = \{a^n b^{2n} \mid n \geq 1\}$.

L_2 = {well-parenthesized strings, with only one kind of parenthesis}; a san example (()()) is well-parenthesized (()())() and (()))(not well-parenthesized.

$L_3 = \{wcw^R \mid w \in \{a,b\}^+\}$.

Solution to point 1.

A DPDA P_1 that recognizes L_1 could have the following working process: in the first initial state q_0 the automaton stacks two characters for any received *a;* when read the first *b* changes state to q_1 and consumes a symbol on the stack for any *b* in input. When the input is ended, if the number of the *b* was twice the number of the *a*, the symbol Z_0 of stack's end is the last symbol. In that case, the automaton executes an ε-move and goes on the final state q_2. See **Figure 1-20.**

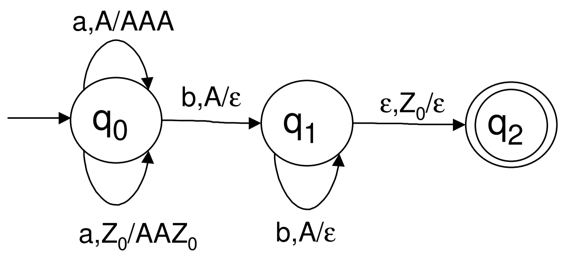


Figure 1-20. DPDA that recognize $L_1 = \{a^n b^{2n} \mid n \geq 1\}$.

One can observe that the first move "a,Z0/AAZ0" is needed because you must read an *a* in the first move, when there is only Z0 on the stack, saving two A on the stack and reinserting Z0 in order to report the end of the stack.

A trace of the working process of the automaton P1 on the given input *abb* is:

<q0, abb, Z0> $\vdash$ <q0, bb, AAZ0> $\vdash$ <q1, b, AZ0> $\vdash$ <q1, ε, Z0> $\vdash$ <q2, ε, ε> and accepts.

A trace of the working process of the automaton P1 on the input a*abbb* is:

<q_0, aabbb, Z_0> $\vdash$ <q_0, abbb, AAZ_0> $\vdash$ <q_0, bbb, $AAAAZ_0$> $\vdash$ <q_1, bb, $AAAZ_0$> $\vdash$ <q_1, b, AAZ_0> $\vdash$ <q_1, e, AZ_0> and stops without accepting.

Note that it is possible to build a DPDA that recognize L_1 without using ε-moves. You just need to memorize the first 'a' read writing a special symbol on the stack (as an example 'B'): so when we will read a 'b' as an input, having 'B' on top of the stack, we will know that the 'b' read corresponds to the first 'a' read. This strategy is realized by the automaton represented in the figure below: **Figure 1-21**.

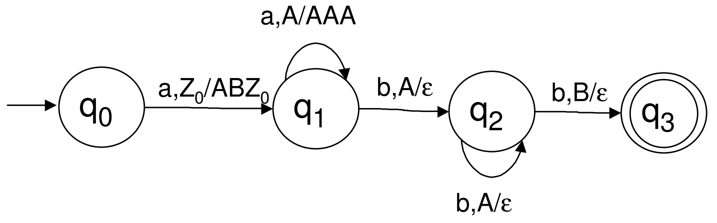


Figure 1-21. Another DPDA that recognizes $L_1 = \{a^n b^{2n} \mid n \geq 1\}$.

Solution to point 2.

The well-formed parenthesis does not require only that the number of the open parenthesis in a word is equal to the closed one, but also that for each word prefix, the number of closed brackets is less than the number of open one (ie you can not close a parenthesis that has not been previously opened).

A DPDA P2 for L2 language can be obtained by saving on the stack symbol A for each open parenthesis and eliminating A from the stack for each closed parenthesis. The automaton must stop without accepting when the current input symbol is a closed parenthesis and there is no A on the stack. Acceptance may be given by an ε-move that leads to a final state q2 when Z0 is the only symbol on the stack. If you want to maintain the condition of determinism, the first move of the automaton must make a transition to a state q1 if input is an open parenthesis with Z0 on the stack. Thus the ε-move for acceptance can be defined by q1 to q2, and not be in "competition" with the first move of the automaton.

See the **Figure 1-22**.

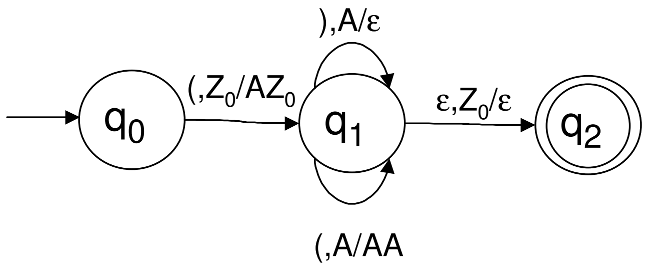


Figure 1-22 DPDA recognizing L2, the language of strings well-formed parenthesis.

A trace of the functioning of the automaton on the (()()) expression is:

<q0, (()()), Z0> ⊢ <q1, ()()), AZ0> ⊢<q1,)()), AAZ0> ⊢<q1, ()), AZ0> ⊢<q1,)), AAZ0> ⊢<q1,), AZ0> ⊢<q1, ε, Z0> ⊢<q2, ε, ε> and accept.

In any configuration on the stack is stored the number of brackets are still open, for which, has yet to be read a closing parenthesis. The stack can therefore open an unlimited number of brackets, allowing you to handle any level of nesting of expressions.

Solution point 3.

An automaton to recognize L3 can save the input string on the stack until the arrival of a *c*; From that moment on, just compare, symbol by symbol, the input to the stack. Indeed, the policy of the LIFO stack saved the string w can be read in reverse, getting exactly w^R. The corresponding automaton is shown in **Figure 1-243**.

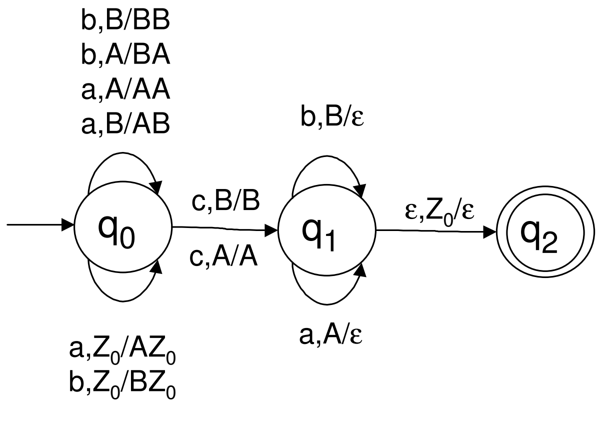


Figure 1-253 DPDA that recognize $L_3 = \{wcw^R \mid w \in \{a,b\}^+\}$.

Exercise 1.2.2

Consider the language of well-formed parentheses (ie where for each open parenthesis corresponding a closed parenthesis) of the arithmetic expressions, which appear only transactions with at least two operands. Examples of strings belonging to the language are: "(a + b)", "((a) + (b * c))" e "((a + b))".

Suppose that in addition to regular parentheses '(' and ')', is also available bracket ']', which has the effect of closing all open parentheses up to that point.

Examples of strings belonging to that language are: "(a + b]", "((a) + (b * c]", "(((((((((a + b]", "(a + b)" e "((a) + (b * (c)]".

Examples of strings not belonging to that language are: "(a + b)]", "((a) + (b * (c)])", "(a]]".

Please define the minimum power automaton recognizing this language.

Note: To simplify the writing you can use a single symbol to represent all of the terms 'a', 'b', 'c', ..., and a single operator.

Solution

The automaton that accepts the language without ']' is shown in **Figure 1-24**. Note that the first parenthesis is "remembered" by the character 'B', whereas the subsequent ones cause writing an "A" on the stack. In this way we recognize the parenthesis that closes an expression, however complex, without ε-moves.

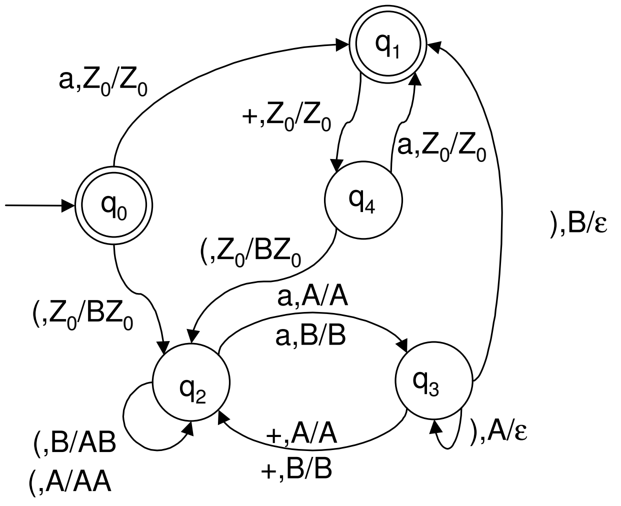


Figure 1-24 DPDA recognizing the language of arithmetic expressions well-formed parenthesis.

The automaton for the language with ']' is a fairly simple extension of the previous one: when one expects a closed parenthesis (ie state q3) it accepts ']', after which it must "forget" all currently open parentheses, which takes place in q3: when on the stack there is only Z_0 it can accept input (moving to q_5) otherwise everything read so far is the operand of an operation (it goes to q_4, where it starts reading another operand).

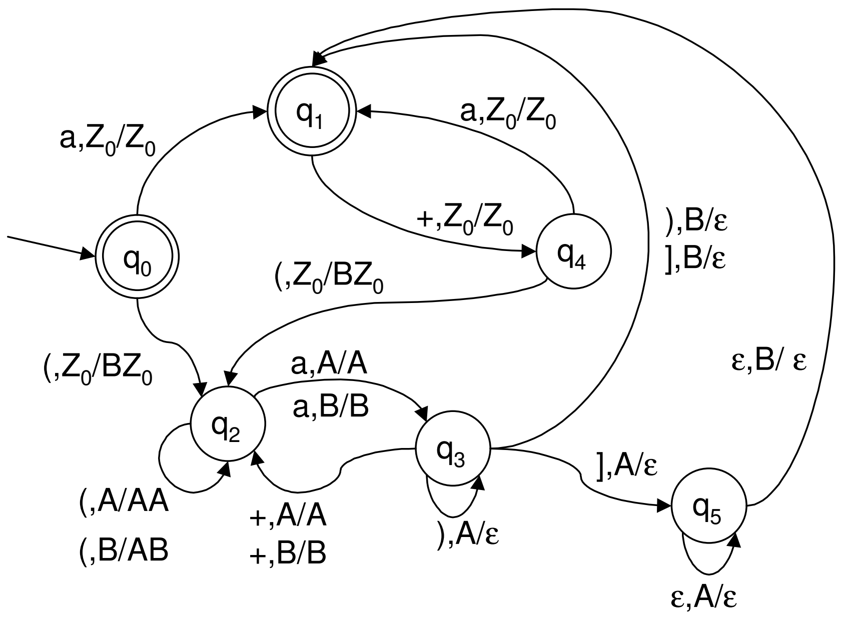


Figure 1-25 DPDA that recognize the language of the of the arithmetic expression well-formed parenthesis

Exercise 1.2.3

Define a deterministic stack automaton that recognizes the following language.

$L = \{ w \mid w \in \{a, b\}^*, \#a(w)=\#b(w) \}$

that is, the set of strings where the number of the ‘a’ characters is equal to the number of the ‘b’ characters.

Solution

The solution is shown in **Figure 1-26**. Every moment, having already marked the string x, the stack save a number of 'A' equal to #a(x)-#b(x), if #a(x)≥#b(x), or a number of 'B' equal to #b(x)-#a(x), if #b(x)≥#a(x). The state q1 is not an acceptance state, but when in q1 the stack is emptied the string x could be accepted, while the automaton goes on state q0 of acceptance with an ε-move.

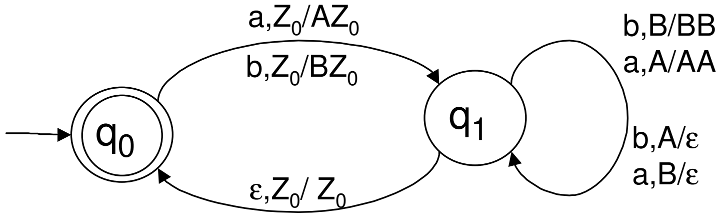


Figure 1-26 Un A DPDA for the language $L = \{ w \mid w \in \{a, b\}^*, \#a(w)=\#b(w) \}$.

Exercise 1.2.4

Define the following deterministic stack automata that recognize the following languages:

$L_1 = \{a^n b^m a^m b^n \mid n, m \geq 1\}$

$L_2 = L_1^*$

$L_3 = \{a^{n1} b^{n1} a^{n2} b^{n2} a^{n3} b^{n3} \ldots a^{nk} b^{nk} \mid k \geq 1, n_i \geq 1, 1 \leq i \leq k\}$

Solution

The automata that recognize L_1, L_2 ed L_3 are shown respectively in **Figure 1-27** and following.

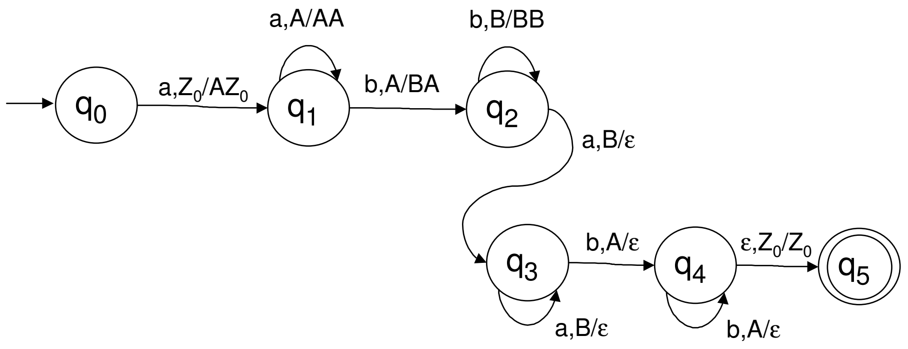


Figure 1-27 A DPDA that recognize $L_1 = \{a^n b^m a^m b^n \mid n, m \geq 1\}$.

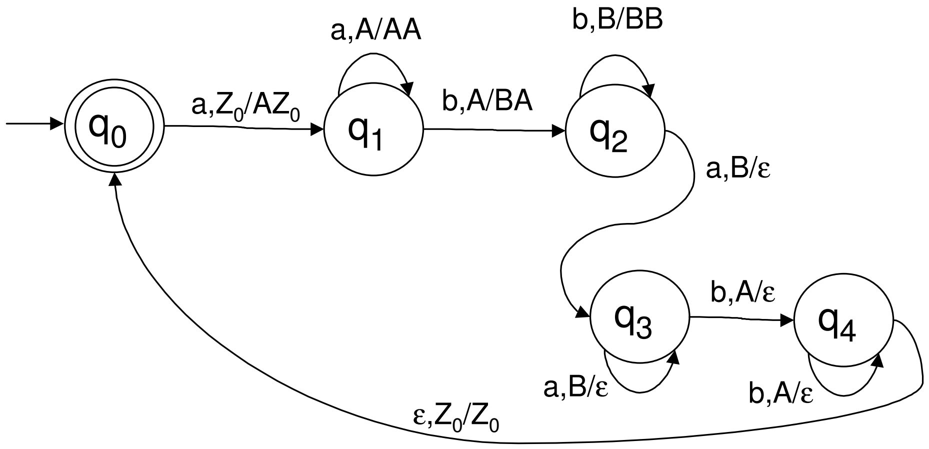


Figure 1-28 A DPDA that recognize $L_2 = \{a^n b^m a^m b^n \mid n, m \geq 1\}^*$.

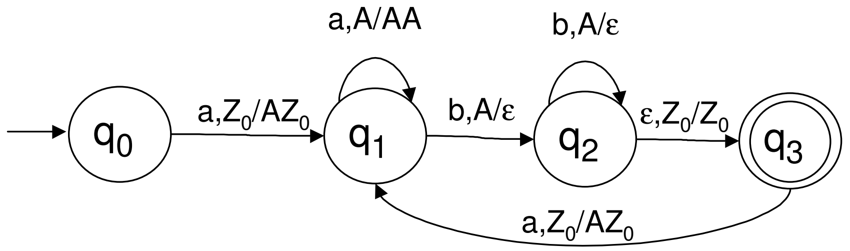


Figure 1-29 A DPDA that recognize $L_3 = \{a^{n1}b^{n1}a^{n2}b^{n2}a^{n3}b^{n3} \ldots a^{nk}b^{nk} | k \geq 1, n_i \geq 1, 1 \leq i \leq k \}$.

Exercise 1.2.5

Build the deterministic pushdown transducer that computes the transduction τ defined by: $\tau(a^n b^m c^m) = d^{3m} e^n$, with $m \geq 1$, $n \geq 1$.

Solution

A simple transducer to compute τ is described in **Figure 1-30**.

The transducer stores on the stack the group of *a*'s without producing anything; then it reads the *b*'s and produces two *d* for each *b* read, storing at each time a *B* symbol on the stack; then it reads the *c*'s, producing at each step another *d*, and comparing their number with the number of B on the stack; eventually by means of a sequence of ε-moves it empties the stack producing at each step an *e*.

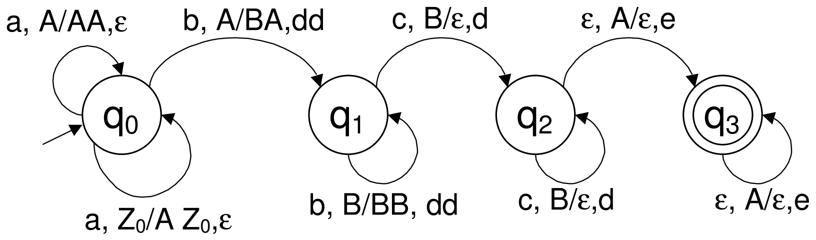


Figure 1-31 Un DPDT che effettua la trasduzione $\tau(a^n b^m c^m) = d^{3m} e^n$, con $m \geq 1$, $n \geq 1$.

An outline of the automaton working process on the string *aabc* is:

<q0, aabc, Z0, ε> $\vdash$ <q0, abc, AZ0, ε> $\vdash$ <q0, bc, AAZ0, ε> $\vdash$ <q1, c, BAAZ0, dd> $\vdash$ <q2, ε, AAZ0, ddd> $\vdash$ <q3, ε, AZ0, ddde> $\vdash$ <q3, ε, Z0, dddee> and accepts.

Note that the precise definition of transduction is essential so that the automaton behaves correctly. Indeed, the automaton can execute an ε-move when it is in the final state with empty input. The transduction is however defined univocally thanks to the condition that states that a configuration for a transducer is final when it is in a final state with empty input, and there are no more possible ε–moves. In the example, when the transducer comes in q3, the underlying recognizer automaton is already in a final configuration, while the transducer does not, because an ε-move is still possible. When also this move is executed, it is no more possible any other ε-move (Z0 is the only symbol left on the stack): the transducer stops and accepts, producing the string *dddee*.

If the input string would be *aab* (that does not belong to the language accepted by the underlying automaton), an outline of the automaton working process would be:

<q0, aab, Z0, ε> $\vdash$<q0, ab, AZ0, ε> $\vdash$<q0, b, AAZ0, ε> $\vdash$<q1, ε, BAAZ0, dd> and it stops without producing anything. In this case the transduction assumes the indefinite value (the string *dd* originated during the execution is ignored), because the transduction definition which assumes that no value is produced when the input string is not recognized by the underlying automaton.

Exercise 1.2.6

Translate a string *x* of the kind *wd*, w∈{*a*, *b*}*, in w^R, which means to translate it in the reflected string. Formally prove that the following pushdown transducer T is able to accomplish this work.

$T = <\{q_o, q_1, q_2\}, \{a, b, d\}, \{A, B, Z_o\}, \delta, q_o, \{q_2\}, \{a, b\}, \eta>$

where δ and η are shown in **Figure 1-32**.

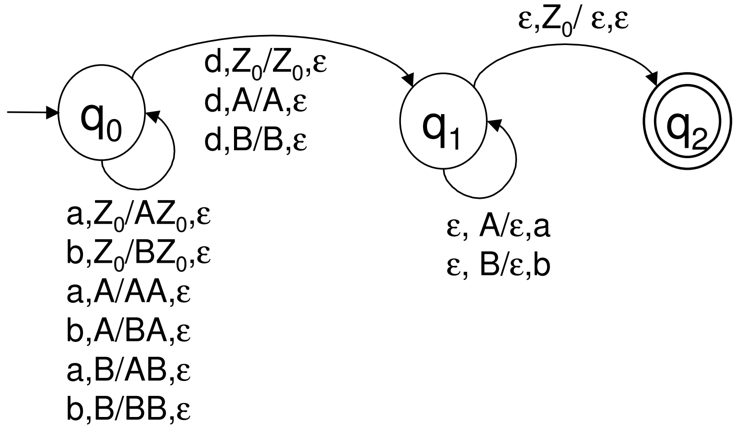


Figure 1-33 Un A DPDT that computes the transduction $\tau(wd) = w^R$.

Solution

The proof is described in the following steps.

1. $<q_o, w, Z_o, \varepsilon> \vdash^* <q_o, \varepsilon, W^R Z_o, \varepsilon>$, where, for each w∈$\{a, b\}^*$, W specifies the result of the substitution of all the lower characters of w with the corresponding uppercases, and W^R specifies, as usual, the reflection of W.
2. $<q_o, d, XZ_o, \varepsilon> \vdash <q_1, \varepsilon, XZ_o, \varepsilon>$ this property derives immediately from the functions δ and η, as shown in the diagram for the transaction from q_o to q_1.
3. $<q1, \varepsilon, XZo, \varepsilon> \vdash^* <q1, \varepsilon, Zo, x>$, where, for each X∈$\{A, B\}^*$, x specify the resulting string from the substitution of the uppercase characters of X with the lowercase ones.
4. $<q_1, \varepsilon, Z_o, x> \vdash <q_2, \varepsilon, \varepsilon, x>$ this property derives directly from the unique transaction between states q_1 and q_2.

The prove of point 1. could be done for induction on the length |w| of the string w.

The induction starts with: having |w|= 0, thus w = ε, and then $W^R = \varepsilon$. We have $<q_o, \varepsilon, Z_o, \varepsilon> \vdash^0 <q_o, \varepsilon, Z_o, \varepsilon>$ where $\vdash^0$ describes a sequence of transactions with zero steps.

Induction step: having |w| = k + 1, k ≥ 0, thus w = xc, $x \in \{a,b\}^*$, $c \in \{a,b\}$, and then $W^R = CX^R$, where C specify the uppercase version of the symbol c. We have $<q_o, xc, Z_o, \varepsilon> \vdash^*$ [for the induction hypothesis] $<q_o, c, X^R Z_o, \varepsilon> \vdash$ [with one of transactions from q_o a q_o] $<q_o, \varepsilon, CX^R Z_o, \varepsilon> = <q_o, \varepsilon, W^R Z_o, \varepsilon>$.

The prove of point 3. is similar.

The induction starts having |X|=0, thus X=ε and then x=ε. One obtains $<q_1, \varepsilon, Z_o, \varepsilon> \vdash^0 <q_1, \varepsilon, Z_o, \varepsilon>$.

Induction step. Having |X|=k+1, k≥0, thus X=WC and then x=wc. We have the following transitions. $<q_1, \varepsilon, WCZ_o, \varepsilon> \vdash^*$ [for the induction hypothesis] $<q_1, \varepsilon, CZ_o, w> \vdash$ [with one of the transactions from q_1 to q_1] $<q_1, \varepsilon, Z_o, wc> = <q_1, \varepsilon, Z_o, x>$.

Exercise 1.2.7

Build deterministic pushdown automata that recognize the following languages:

$L_1 = \{a^n b^n \mid n \geq 1\}^{\wedge} \cdot \$$	(The complement operator ^ is assumed to be applied to the universal language {a,b}*, so that the special character $ is an indicator of the end of the string.)
$L_2 = \{a^n b^n \mid n \geq 1\}^{\wedge}$	(The complement operator ^ is assumed to be applied on the universal language {a,b}*)

Solution to point 1

There are many ways to build the automaton, and there are many automata that can recognize L_1. A simple way is to build the automaton that recognize $L = \{a^n b^n \mid n \geq 1\}\$$ at first: see **Figure 1-34**.

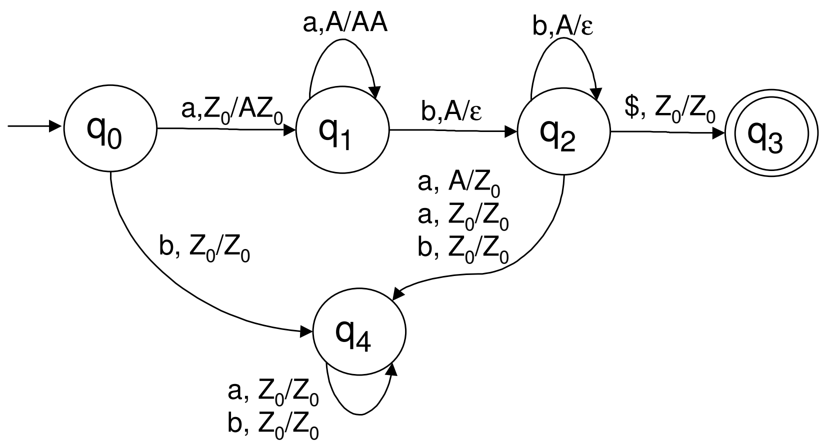


Figure 1-35 Pushdown automaton that recognize L = $\{a^nb^n \mid n \geq 1\}\$$.

It is easy to turn the automaton in **Figure 1-36** into the automaton that recognize L_1.One have to:

- Add an edge from each non terminal state, going to a new state q_5 when a '$' character is read and
- Transform q_3 in a non final state.

The automaton described is shown in **Figure 1-37**.

.

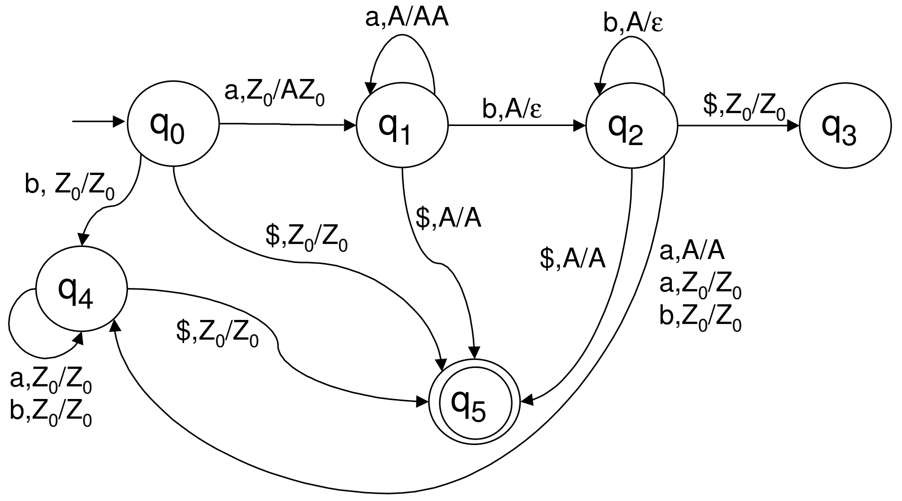


Figure 1-38 Deterministic Stack Automaton that recognize $L_1 = \{a^nb^n \mid n \geq 1\}^{\wedge}\$$.

Solution to point 2

In a similar way, to build the automaton that recognize L_2, we can start from the automaton that recognize L = $\{a^nb^n \mid n \geq 1\}$ (notice the absence of the ending character '$'). The deterministic pushdown automaton that recognize L is reported in **Figure 1-39**. In this automaton the first *a* read is stored putting an 'X' character on the stack, while the next *a*'s will be stored putting an 'A' character.

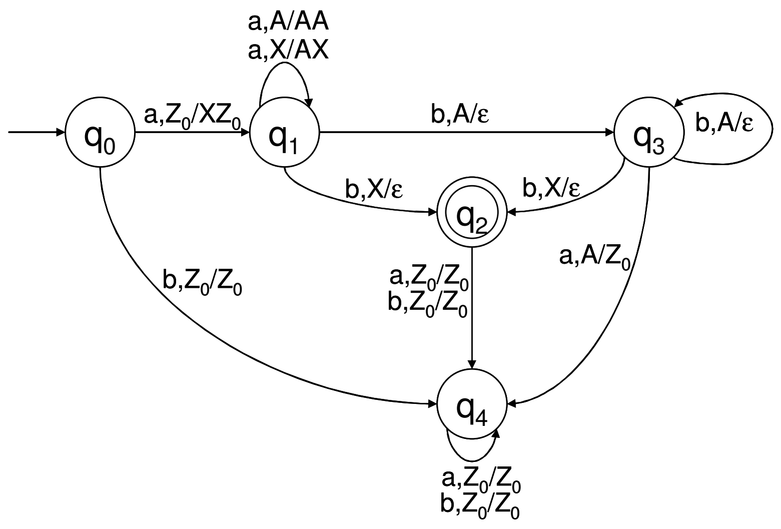


Figure 1-40 pushdown automaton that recognize L = $\{a^nb^n \mid n \geq 1\}$.

The automaton that recognize L_2 is realized starting from the automaton in **Figure 1-41** simply complementing final and non final state. The result is shown in **Figure 1-42**.

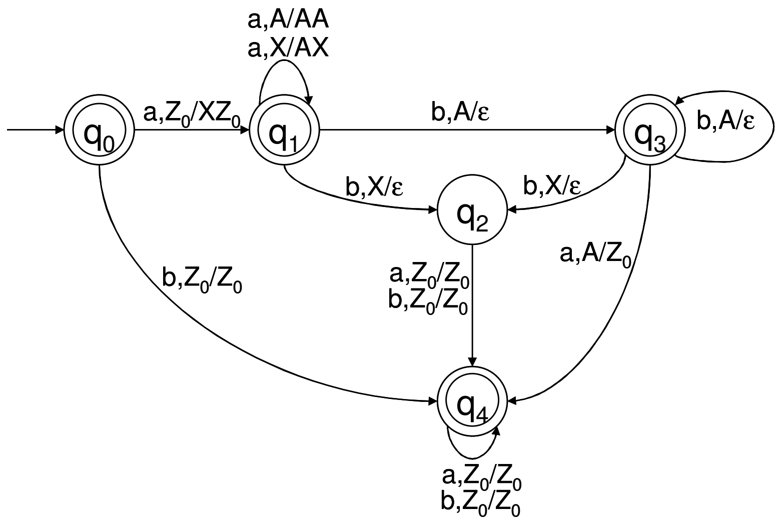


Figure 1-43 Deterministic stack automaton that recognize $L_2 = \{a^nb^n \mid n \geq 1\}^\wedge$.

Exercise 1.2.8

Build the pushdown automaton that recognizes the language L = $\{a^mb^nc^pd^ne^5 \mid m, n, p \geq 1\}$. It is *preferable but not necessary* to use 7 states at most.

Solution

It is rather easy to build a DPDA with 8 states that recognizes the given language. Such automaton is shown in **Figure 1-44**. Observe that all the 'a''s and the 'b''s are read in the state q_0. When the first 'a' is read, an 'A' is written on the stack, in order to enable the reading of 'b'; indeed, being m ≥ 1, we must read at least an 'a' before to read the 'b''s.

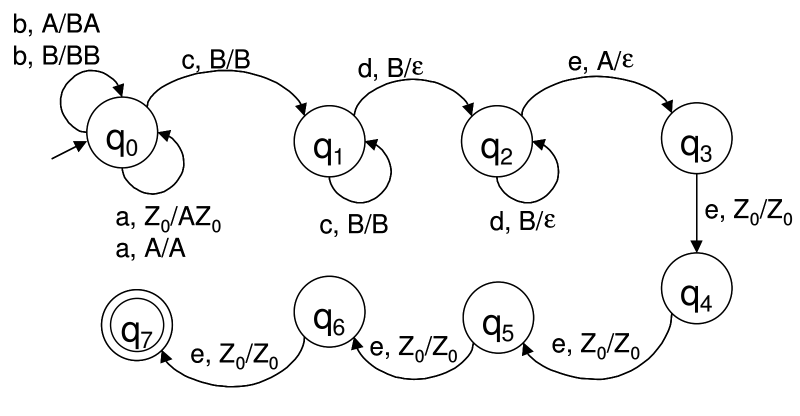


Figure 1-45 Deterministic pushdown automaton that recognize L= {$a^m b^n c^p d^n e^5$ | m, n, p ≥ 1}.

To reduce the number of states of the automaton we can use the 'A' placed on the stack after the first 'a' is read. Indeed we can use four states to count four 'e', and the 'A' on the stack to count the fifth 'e'. The DPDA that exploit this principle is shown in **Figure 1-46**.

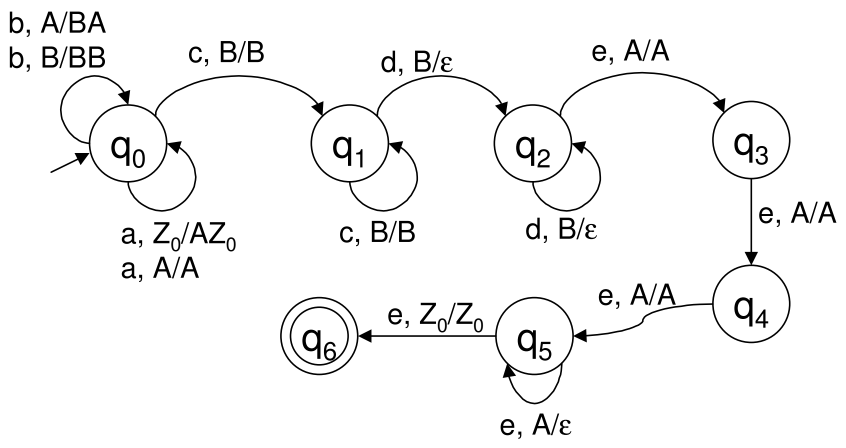


Figure 1-47 DPDA with 7 states that recognize L.

1.3 Turing Machines

Exercise 1.3.1

Build a TM to recognize the following languages:

$L_1 = \{wcw \mid w \in \{a,b\}^+\}$;

$L_2 = \{a^n b^{n^2} \mid n \geq 1\}$

Solution to point 1.

Let us define a one memory tape TM, with Q = {q_0, q_1, q_2, q_3}. The automaton stores on the memory tape the content of the input tape until it reads the first *c* character, then it comes back at the beginning of the memory tape and it compares its content with the remaining input. See **Figure 1-48**. One must pay attention that the only way to come back at the beginning of the input tape is rewrite one by one all the cells, moving to the left: the TM definition we consider must indeed rewrite at each move the content of the cell under the tape head.

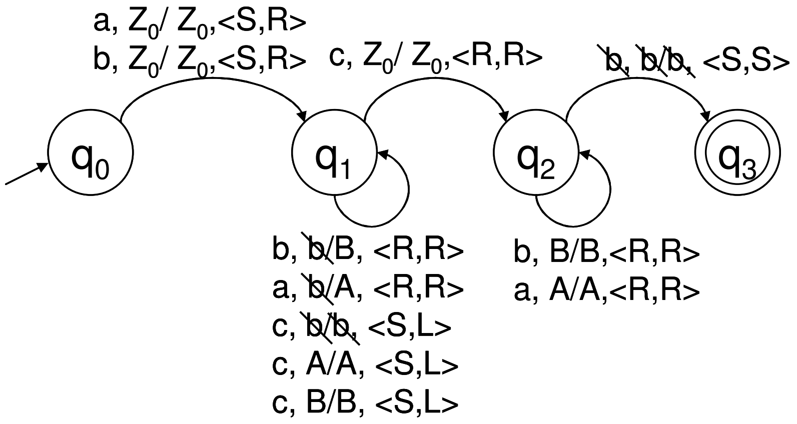


Figure 1-49 A one memory tape TM that recognizes $L_1 = \{wcw \mid w \in \{a,b\}^+\}$.

Observe that there are not outgoing transitions to the final state, as required by the TM definition. A trace of the machine behavior on the string abcab is:

$<q_0, \uparrow abcab, \uparrow Z_0> \vdash <q_1, \uparrow abcab, Z_0\uparrow> \vdash <q_1, a\uparrow bcab, Z_0A\uparrow> \vdash <q_1, ab\uparrow cab, Z_0AB\uparrow> \vdash <q_1, ab\uparrow cab, Z_0A\uparrow B> \vdash <q_1, ab\uparrow cab, Z_0\uparrow AB> \vdash <q_1, ab\uparrow cab, \uparrow Z_0AB> \vdash <q_2, abc\uparrow ab, Z_0\uparrow AB> \vdash <q_2, abca\uparrow b, Z_0A\uparrow B> \vdash <q_2, abcab\uparrow, Z_0AB\uparrow> \vdash <q_3, abcab\uparrow, Z_0AB\uparrow>$ and accepts.

Solution to point 2.

Let us use a two memory tape TM M. For each *a* in input it writes an *A* on both the memory taspes. The first tape is used ad counter to read exactly n *b*, the second tape is used as counter to execute n times the read of n *b* operation. In q_1 M performs the read of n *b*, moving one position left on the first tape and standing on the second tape at each step. At this point M removes an *A* from the second tape (transition from q_1 to q_2), it comes back at the beginning of the first memory tape (selfloop in q_2) and comes back in q_1. When M is the the state q_1 and there are no more *b*'s in input, M checks to be at the beginning of the first tape and goes in q_2 removing an *A* from the second tape. Now, if it is at the beginning of the second tape too, it goes in q_3 and accepts, because are been read exactly n^2 *b*.

See **Figure 1-50**.

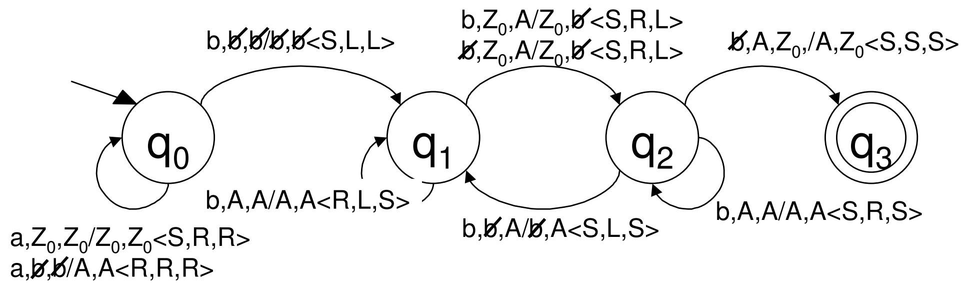


Figure 1-51 A two memory tapes TM for $\{a^n b^{n^2} \mid n \geq 1\}$.

Exercise 1.3.2

Define Turing machines (with custom tapes) that recognize the following languages.

$L_1 = \{ wcw^R \mid w \in \{a, b\}^+ \}$

$L_2 = \{ x \in \{a, b, c\}^* \mid \#a(x)=\#b(x) \text{ or } \#a(x)=\#c(x) \}$

(where, for a character α and a string *x*, the expression $\#\alpha(x)$ represent the number of times that the character α appears on the string *x*.

Solution

For the language L_1 we use a TM with one tape, shown in **Figure 1-52**, used exactly as a stack where we store w^R, which is after compared character by character with the input part that follow the 'c'.

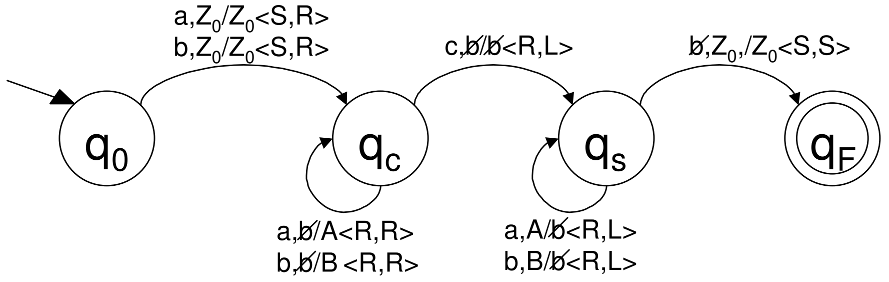


Figure 1-53 A TM for $L_1 = \{ wcw^R \mid w \in \{a, b\}^+ \}$.

For the language L_2 we use a TM with four tapes, shown in **Figure 1-54**, to be uses as unit counter, using the symbol '*'.

T_1 stores #a, to compare with #b;

T_2 stores #a, to compare with #c;

T_3 stores #b;

T_4 stores #c.

First of all the machine scans the input tape, storing the number of character on the three kind of tapes. After that, with a couple of ε-moves, compare the numbers in the following way: until possible, delete at the same time a '*' from T_3 and from T_1, and from T_4 and from T_2. At the end of this phase, we accept if and only if at least one of the two pairs, T_1 and T_3 or T_2 e T_4, are empty.

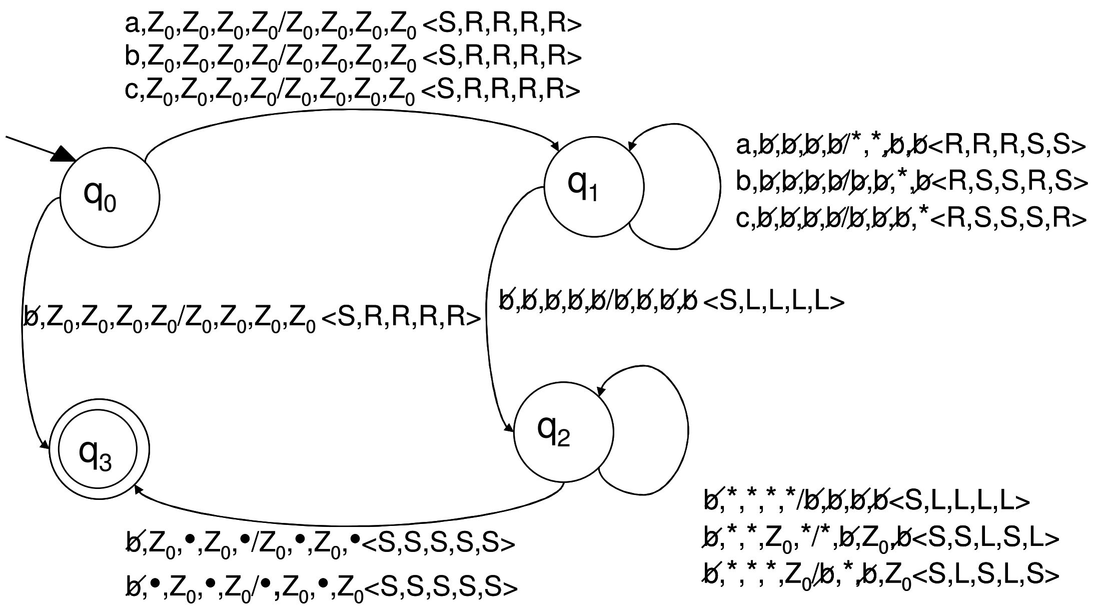

a,Z0,Z0,Z0,Z0/Z0,Z0,Z0,Z0 <S,R,R,R,R>
b,Z0,Z0,Z0,Z0/Z0,Z0,Z0,Z0 <S,R,R,R,R>
c,Z0,Z0,Z0,Z0/Z0,Z0,Z0,Z0 <S,R,R,R,R>
q0
q1
a,b̸,b̸,b̸,b̸/*,*,b̸,b̸<R,R,R,S,S>
b,b̸,b̸,b̸,b̸/b̸,b̸,*,b̸<R,S,S,R,S>
c,b̸,b̸,b̸,b̸/b̸,b̸,b̸,*<R,S,S,S,R>
b̸,Z0,Z0,Z0,Z0/Z0,Z0,Z0,Z0 <S,R,R,R,R>
b̸,b̸,b̸,b̸,b̸/b̸,b̸,b̸,b̸ <S,L,L,L,L>
q2
q3
b̸,*,*,*,*/b̸,b̸,b̸,b̸<S,L,L,L,L>
b̸,*,*,Z0,*/*,b̸,Z0,b̸<S,S,L,S,L>
b̸,*,*,*,Z0/b̸,*,b̸,Z0<S,L,S,L,S>
b̸,Z0,•,Z0,•/Z0,•,Z0,•<S,S,S,S,S>
b̸,•,Z0,•,Z0/•,Z0,•,Z0<S,S,S,S,S>

Figure 1-55 A TM for $L_2 = \{ x \in \{a, b, c\}^* \mid \#a(x)=\#b(x) \text{ or } \#a(x)=\#c(x) \}$.

In the transition graph, the self loops on q_1 and the transition to the final state q_F are commented in order to indicate which features of the input string correspond every moves; Furthermore we adopted the convention to indicate with the symbol '•' any element of the alphabet.

Exercise 1.3.3

Build an abstract machine that accepts the following language:

$L = \{wcw \mid w \in \{a,b\}^*\} \cup \{wcw^R \mid w \in \{a,b\}^*\}$

The machine can be freely chosen among those known in the literature or can be specifically defined by adapting its operation to the specific needs of the problem. But it must still be deterministic. It means that in any configuration it cannot perform two different moves by reading a single character.

Solution

Obviously, language consists of two components that require recognitional mechanisms between their duals: you need a queue structure to recognize strings of type wcw and a pushdown structure to recognize strings of type wcw^R.

If we want to recognize L by a traditional abstract machine, we can use a two-tapes Turing machine: the first tape will act as a queue, the second tape will act as a pushdow:

- The machine reads the string w until detecting the first ‘c’ character. The machine reads the first string w to the identification of the character 'c'. While reading copy w on both tapes.
- After reading 'c' of the first tape, called a "queue", the head come back to its beginning, while the second tape, called a "pushdown" is located on the last character of w
- After that, it starts reading the string after ‘c’. During this reading both heads of the two tapes make a move in a way to check the head of the queue if the string is a w or the head of the pushdown if the string is w^R.
- If one of the heads does not find matching character, it stops and the comparison continues by the other one.
- The machine accepts if at least one of the head reaches the bottom of the comparison.

Exercise 1.3.4

A single tape Turing machine, shown in **Figure 1-56**, has only one tape used simultaneously as input, output and memory. The tape is infinite in both directions and the control unit can perform different moves depending on the symbol under the head and the internal state. A move consists in:

- change of the internal state;

- rewrite the symbol under the head;

- moving the head to the left (L), right (R) or in no direction (S).

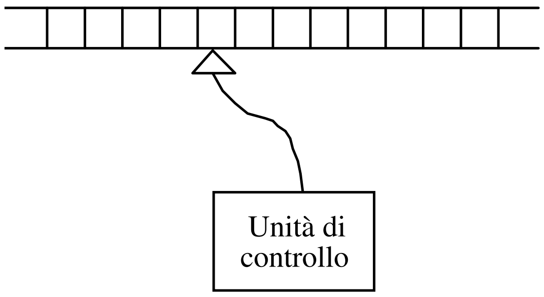


Figure 1-57 A single-tape TM infinite in both directions.

Conventionally the entrance of the machine is contained in the tape at the beginning of computation, with the head positioned on the leftmost symbol is not empty. When the computation stops, the non-empty portion of the tape is considered the output of the machine.

Formalize the single-tape Turing machine, as outlined above, and in particular by defining the terms configuration, initial configuration and final transition, recognition of language and translation of a language.

Solution

Formalize the single tape TM as a quintuple <Q, A, δ, q_o, F>, where

Q is the finite set of states;

A is the alphabet of the tape, which includes the special symbol empty b,/ ;

$q_0 \in Q$ is the initial state

$F \subseteq Q$, is the set of final states;

δ is the transition function

$$\delta: (Q - F) \times A \rightarrow Q \times A \times \{L, R, S\}$$

Define as configuration of a single-tape TM a tuple like

$c = <q, x \uparrow i\ v>$, con $x, v \in A^*, i \in A, q \in Q$,

represented for brevity with the string "xqiv", which describes a single-tape TM that is in state q with the tape in the situation shown in figure Figure 1-61, with the portion of the tape consists of the empty string xiv and the head placed on the cell that contains the symbol *i*. In the initial configuration the state *q* is the initial q_0 e $x = \varepsilon$; in a final configuration $q \in F$.

Represent graphically the function of transiction δ specifying the value $\delta(q, i) = <p, c, N>$ as shown in **Figure 1-58**.

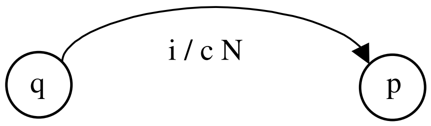


Figure 1-59 A generic arc of the graph of transitions of a single-tape TM.

The binary relation '$\vdash$' of transiction between configurations is defined as follow: x q i v $\vdash$ x' q' i' v' if and only if $\delta(q, i) = <p, c, N>$, q'=p, and is, exclusively, one of three cases (note that the strings x and v may or may not be empty, $x = \underline{x}\ \underline{r}$ or $x = \varepsilon$, and $v = \underline{s}\ \underline{v}$ or $v = \varepsilon$).

1. N = S, x'=x, v'=v, i'=c.

Configurations before and after a transition of this type are shown in **Figure 1-60**.

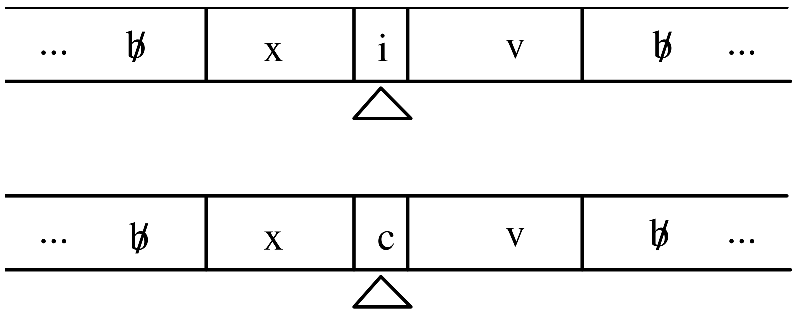


Figure 1-61 Transition with head stopped.

2. N = R,x' = x c and

if $v = \underline{s}\ \underline{v}$, $v' = \underline{v}$, $i' = \underline{s}$ (see the **Figure 1-62**);

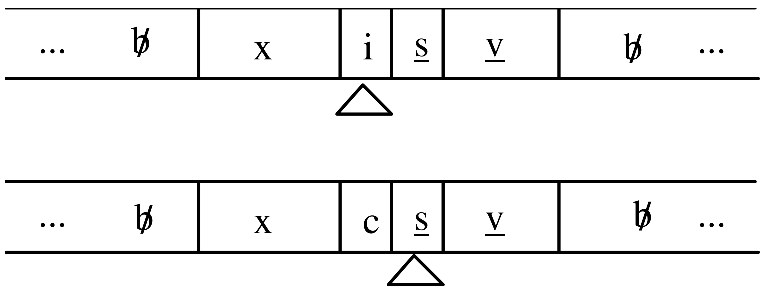


Figure 1-63 Transition with head moving to the right.

if v = ε, v' = ε, i' =b,/ (see the **Figure 1-64**).

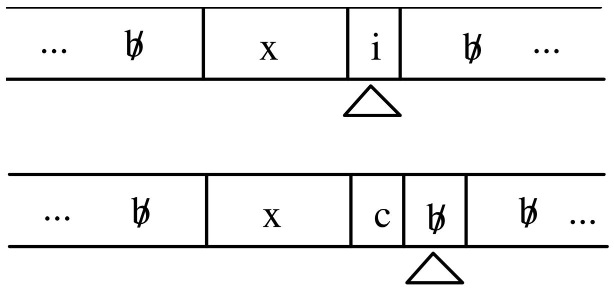


Figure 1-65 Transition with head moving to the right on a b,/ .

.

3. N = L, v' = c v and

if x = x̲ r̲, x' = x̲, i' = r̲ (see the **Figure 1-66**);

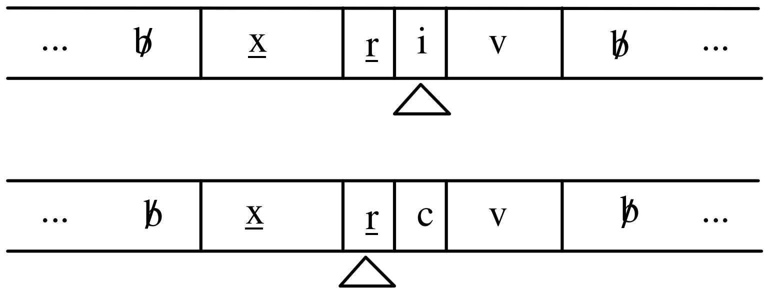


Figure 1-67 Transition with head moving left.

if x = ε, x' = ε, i' =b,/ (see the **Figure 1-68**)

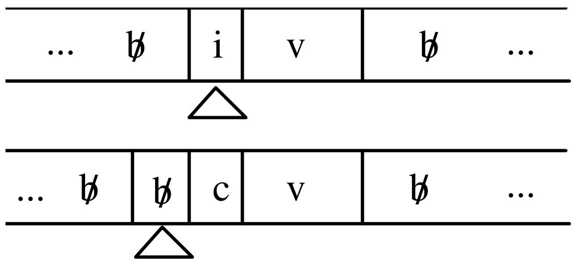


Figure 1-69 Transition with head moving left on a b,/ .

Exercise 1.3.5

Formalize the definition of single-tape TM, finite to left and infinite to right, and describe a TM of this kind to recognize the language L = $\{a^nb^n \mid n \geq 1\}$.

Solution

A single-tape TM M finite to left, may be described as a sixfold <Q, A, I, δ, q0, F>, with Q a finite set of state, F Q is the set of final states, A is a finite alphabet, called the tape alphabet including at least the blank symbol b /, I A is the input alphabet, q0 Q is the initial state, while the transition function δ, or partial, is defined as δ :

$(Q\text{-}F) \times A \to Q \times A \times \{R,L,S\}$.

A configuration is a pair <q, α1↑α2>, with q Q, α1, α2 A *.

An initial configuration is a configuration like this: <q0, ↑α> with α I *.

Move: let c = <q, α1↑aα2> configuration, where the configuration is of type c = <q, α↑>, let taken *a* as symbol of blank b /, let also δ (q, a) = <q', a', N>. Then:

if N=S we have c $\vdash_M$ <q', α_1↑a'α_2>;

if N=R we have c $\vdash_M$ <q', α_1a'↑α_2>;

if N=L we have c $\vdash_M$ <q', β_1↑ba'α_2>, with $\alpha_1 = \beta_1 b \neq \varepsilon$ (if $\alpha_1 = \varepsilon$, the move is not defined).

The *accepted language* from single-tape TM M is L(M) = {x|x ∈ I^* and <q_0, ↑x> $\vdash^*_M$ <q, α_1↑α_2> with q ∈ F}.

A TM M for L = $\{a^nb^n \mid n \geq 1\}$ for example, may delete from a tape *a* and *b* each time like this. M replaces the more left *a* with an X, it moves to the leftmost *b* and replaces it with a Y. If it finds a blank, M stops without accepting (there is at least one more *a* than *b*). M then search the symbol rightmost X, it moves right one cell and, if the content is still *a*, it repeats the loop. If there are no more left *a*, M still looking for *b*, if so, M accepts, otherwise it stops accepting.

Formally, we have Q = $\{q_0, q_1, q_2, q_3, q_4\}$, A = {a, b , X, Y, b,/ }, F = $\{q_4\}$, I = {a,b}, while transition function δ is defined in **Figure 1-70**.

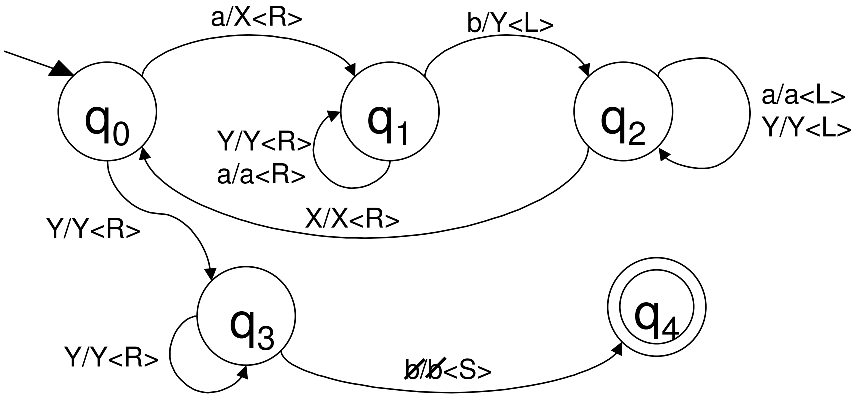


Figure 1-71 A single-tape TM for L = {a^nb^n | n≥1}.

A possible operation on the string *aabb* is:
<q_0, ↑aabb> ⊢<q_1, X↑abb> ⊢<q_1, Xa↑bb> ⊢<q_2, X↑aYb> ⊢ <q_2, ↑XaYb> ⊢ <q_0, X↑aYb> ⊢ <q_1, XX↑Yb> ⊢ <q_1, XXY↑b> ⊢ <q_2, XX↑YY> ⊢<q_2, X↑XYY> ⊢<q_0, XX↑YY> ⊢<q_3, XXY↑Y> ⊢<q_3, XXYY↑> ⊢<q_4, XXYY↑> and accepts.

Exercise 1.3.6

Define a single-tape TM which recognizes the language L = {ww^R | w∈{a, b}*}

Solution

The solution showed in **Figure 1-72**, where the graph of transitions is displayed with the conventions suggested in Exercise 1.3.4, runs the input string alternately from left to right and vice versa, deleting letters equal to extremes, until the tape is blank.

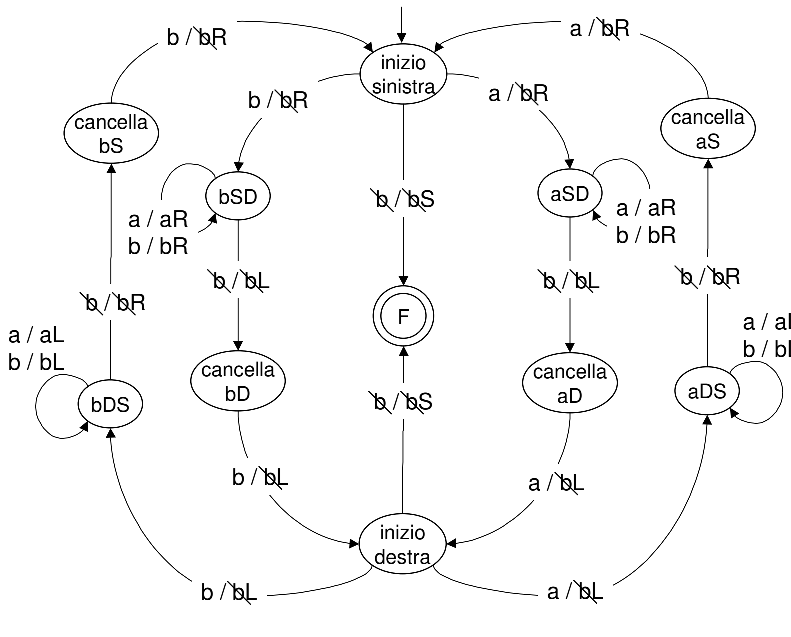


Figure 1-73 A single-tape TM for L = {ww^R | w∈{a, b}*}.

Exercise 1.3.7

A TM that generates language is a TM without input tape, but with k tapes of memory and one output tape. The TM starts with the empty tape output and writes from left to right, strings separated by #. Words can be repeated nor is there any particular order in which they are generated. TM never ends to compute even when the language is finite (there are no final states). The language generated by a TM of this type is the set of strings that are written during the execution between two characters # on the output tape.

1. Formally define the generating TM.

2. Describe a TM that generates language $L = \{a^n b^n \mid n \geq 1\}$.

Solution to point 1.

For simplicity, we consider only a TM with tape storage, but the extension with k tapes is immediate.A TM that generates language is a 7-tuple M = <Q, Γ, O, S, δ, η, q_0, Z_0>, with the usual meaning of symbols. Z0 and b, / belong to Γ, # □ O. The function δ, maybe partial, is defined as:

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R,L,S\}$.

The function η, is defined where δ is, is:

$\eta : Q \times \Gamma \rightarrow O \times \{R,S\}$

A configuration is a triple<q, $\alpha\uparrow\beta$, $u\uparrow o$>, with $q \in Q$, $\alpha, \beta \in \Gamma^*$, $u \in O^*$, $o \in O$.

An initial configuration is a configuration like this: <q_0, $\uparrow\alpha$, $\uparrow Z_0$>.

The concept of final configuration is not relevant for this type of machine.

The relation□ ↓ transition is a binary relation between configurations such that:

<q, $\alpha\uparrow\beta$, $u\uparrow o$> $\vdash$ <q', $\alpha'\uparrow\beta'$, $u'\uparrow o'$> if, and only if:

$\delta(q, A) = (q', A', N)$, $\eta(q,a) = (o'', M)$ with:

if N=S then $\alpha' = \alpha A'$, $\beta' = \beta$;

if N=R then $\alpha' = \alpha$, $\beta' = A'\beta$;

if N=L then, in case of $\alpha = \alpha''A''$ we have $\alpha' = \alpha''$, $\beta' = A''A'\beta$;

if M = R then $u' = uo''$, $o' = \not{b}$;

if M = S then $u' = u$, $o' = o''$.

Finally, language generated by M is $L(M) = \{x \mid \langle q_0, \uparrow\alpha, \uparrow Z_0\rangle \vdash^* \langle q, \alpha\uparrow\beta, u\#x\#\uparrow\rangle$, with $\alpha, \beta \in \Gamma^*$, $u \in O^*$, $x \in (O - \{\#\})^*\}$.

Solution to point 2.

A TM for generating L is shown in **Figure 1-74**. After an initialization move from q0 to q1, the TM writes a number of a equal to the number of X on the tape memory. When there are no more X, the TM writes an a in output and simultaneously adds a X on tape memory, going to the state q2. In Q2, the TM writes in output b to every X on the tape memory, finally returning to the beginning of the tape memory. Now it writes a # to indicate that a phrase like $a^n b^n$ was generated and goes to q1, resuming the cycle with n + 1 instead of n.

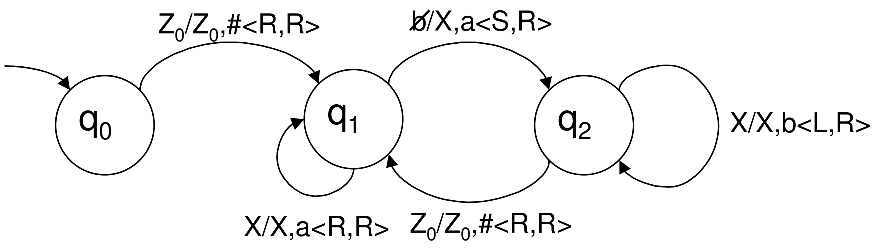


Figure 1-75 A TM to generate $L = \{a^n b^n \mid n \geq 1\}$.

Exercise 1.3.8

1. Build a single tape TM that performs the transduction $\tau(x) = xx^R$, with $x \in \{a,b\}^*$;
2. Build a single tape TM that performs the transduction $\tau(a^n b^m) = a^{n+m}$, $n \geq 0$, $m \geq 1$.

Solution to point 1.

A single memory tape transductor can, at the same time, copy the content of the input tape on the memory tape and give it in output. When the input tape reaches its end, the transductor comes back to the left on the memory tape and gives in output the stored characters. The required TM is shown in **Figure 1-76**.

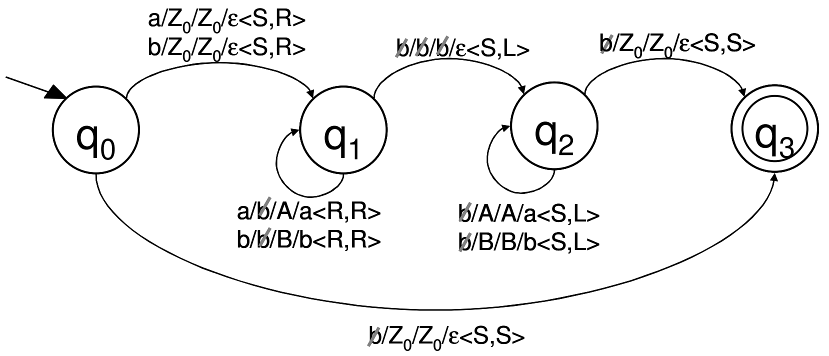


Figure 1-77 Single-tape TM that performs the transduction $\tau(x) = xx^R$, with $x \in \{a,b\}^*$.

Solution to point 2.

The transduction of a string x, for a single-tape TM M, can be defined as the content of the tape when M stops accepting x. More formally, $\tau(x) = y$ if, and only if, $<q_0, \uparrow x> \vdash^*_M <q, y_1 \uparrow y_2>$ with $q \in F$, $y = y_1 y_2$.

The required TM that performs the transduction is shown in **Figure 1-78**. Please note that the shown TM is equivalent to a finite-state transductor.

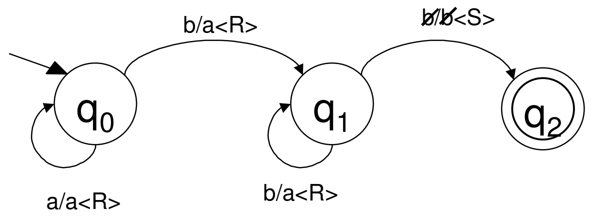


Figure 1-79 A single-tape TM that performs the transduction $\tau(a^nb^m) = a^{n+m}$, $n\geq0$, $m\geq1$.

Exercise 1.3.9

Build a TM that recognizes the complement of the following language:

$L = \{a^nb^{n^2} | n >= 1\}$

Solution

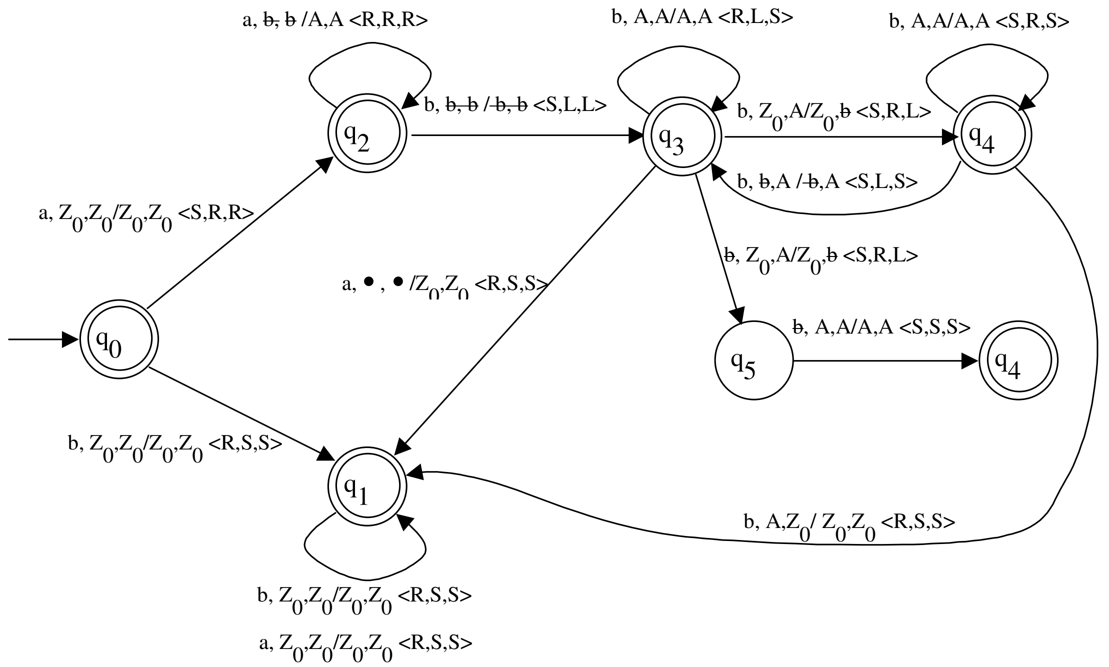


Figure 1-80 The complement of the TM that recognizes the language $L = \{a^nb^{n^2} | n >= 1\}$

A possible solution is shown in **Figure 1-81**. The symbol "•"indicates "any character that is under the reading head". It is therefore a simple syntactic shortcut.

The shown TM is directly derived from the one that recognizes L (and that appears in the solution of the point 2 of the Exercise 1.3.1). The main differences are:

- The state q_1 used to recognize the strings that start with 'b'.
- Only one state (corresponding at the reading of $a^nb^{n^2}$) is not final. This state is reached "precautionally" when at the same time there are no more b's in input and the reading

head is on the beginning of the first tape (i.e. a number of b's multiple of n have been read). If this multiple is less than n^2, a final state (q_6) can be reached. The multiple cannot be greater than n^2 (when the b's are more than n^2 s it exits from q_4 toward q_1). It remains in q_5 (without recognizing the input string) only when the string is $a^nb^{n^2}$.

- There were introduced some transitions toward q_1 that allow to recognize strings $a^nb^{n^2}\{a,b\}^+$, and $a^nb^ma\{a,b\}^*$.

Note:this machine uses the same convention of a finite-state automaton that can pass through "final" states without necessarily stopping.

A solution where is adopted the convention that the final states are only Halt states is shown in **Figure 1-82**. This new machine still reaches an halt state before accepting and it is therefore forced to read the first blank after the input string to "understand" that the string is over.

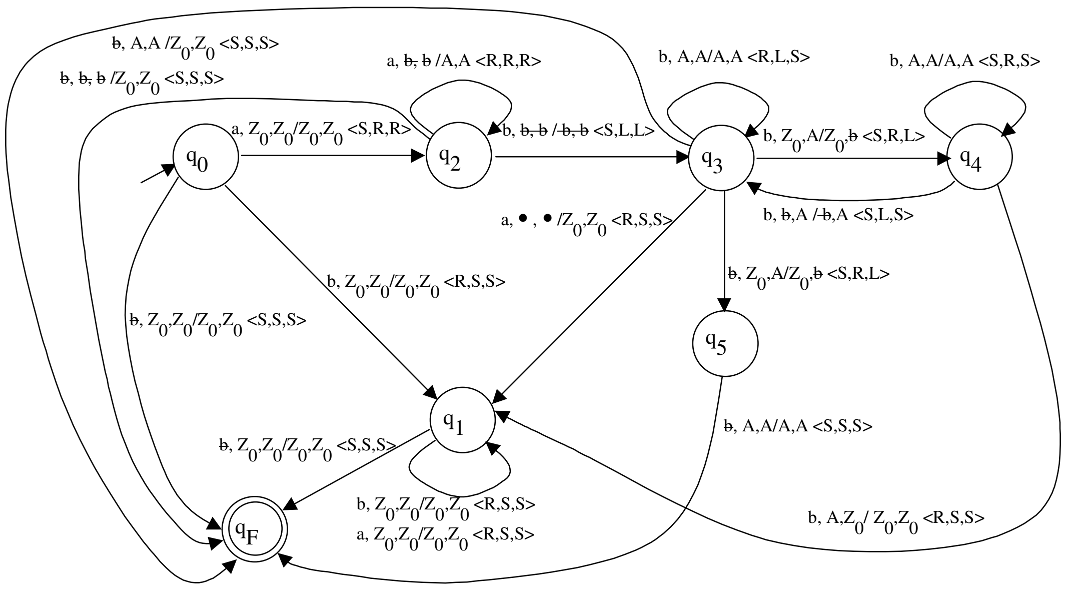


Figure 1-83 TM that recognizes the complement of the language $L = \{a^nb^{n^2} | n >= 1\}$

1.4 Grammars and languages

Exercise 1.4.1

Define a context free grammar for the language of the well-formed parenthesis belonging to the following four categories: '(' and ')', '**do**' and '**od**', '**begin**' and '**end**', '**if**' and '**fi**'.

Solution

$S \rightarrow SS$	$S \rightarrow (S)$	$S \rightarrow$ **do** S **od**
$S \rightarrow \varepsilon$	$S \rightarrow$ **if** S **fi**	$S \rightarrow$ **begin** S **end**

Exercise 1.4.2

Define a context free grammar for the language

$$L = \{a^n b^m a^m b^n \mid n, m \geq 1\}$$

Solution

A possibile set of productions is:

$S \rightarrow a\ P\ b$

$P \rightarrow a\ P\ b \mid b\ R\ a$

$R \rightarrow b\ R\ a \mid \varepsilon$

Another set of productions (equivalent) is:

$S \rightarrow a\ P\ b$

$P \rightarrow a\ P\ b \mid R$

$R \rightarrow b\ R\ a \mid ba$

Exercise 1.4.3

Define a grammar for the language

$$L = \{ w \in \{a, b, c\}^* \mid \#a(w) > \#b(w) > \#c(w) > 0 \}$$

Solution

The grammar can be organized in three sets of productions. The first one generates $A^n B^m C^k$, with n>m>k.

$S \rightarrow A\ S_1$
$S_1 \rightarrow A\ S_1 \mid A\ B\ S_2$
$S_2 \rightarrow A\ B\ S_2 \mid A\ B\ C\ S_3$
$S_3 \rightarrow A\ B\ C\ S_3 \mid \varepsilon$

The second set of productions ensures that starting from a string composed by A, B and C any their permutation can be obtained.

$A\ B \rightarrow B\ A$
$A\ C \rightarrow C\ A$
$B\ C \rightarrow C\ B$
$B\ A \rightarrow A\ B$
$C\ A \rightarrow A\ C$
$C\ B \rightarrow B\ C$

The third set of productions ensures that any non-terminal character can be transformed in the corresponding terminal character.

$A \rightarrow a$
$B \rightarrow b$
$C \rightarrow c$

Exercise 1.4.4

Define a grammar for the language $L = \{ ww \mid w \in \{a, b\}^* \}$

Solution

$V_N = \{S, S_1, A, B, A_1, B_1,]\}$, $V_T = \{a, b\}$

$S \to S_1\ ]$	$S_1 \to a\ S_1\ A$	$S_1 \to b\ S_1\ B$	$S_1 \to \varepsilon$
$B\ ] \to B_1\ ]$	$A\ ] \to A_1\ ]$	$A\ A_1 \to A_1\ A$	$A\ B_1 \to B_1\ A$
$B\ A_1 \to A_1\ B$	$B\ B_1 \to B_1\ B$	$A_1 \to a$	$B_1 \to b$
$] \to \varepsilon$			

The first four productions are used to build strings of the kind $wW^R]$. The subsequent ones transform the non-terminal A or B adjacent to the ‘]’ in the corresponding A_1 or B_1. Such non-terminals “go up” the string on the left side until a similar (A_1 or B_1) non-terminal character is founded. The result is a string of the kind $wW_R]$. At this point the remaining productions transform A_1 or B_1 in *a* and *b* respectively, and eliminate the ‘]’, by now useless. The result are strings of the kind ww, exactly as desired.

For example, *abbaaabbaaa* is obtained by the following derivation:

$S \Rightarrow S_1] \Rightarrow aS_1A] \Rightarrow abS_1BA] \Rightarrow abbS_1BBA] \Rightarrow abbaS_1ABBA] \Rightarrow abbaaS_1AABBA] \Rightarrow abbaaaS_1AAABBA] \Rightarrow abbaaaAAABBA] \Rightarrow abbaaaAAABBA_1] \Rightarrow abbaaaAAABA_1B] \Rightarrow abbaaaAAAA_1BB] \Rightarrow abbaaaAAA_1ABB] \Rightarrow abbaaaAA_1AABB] \Rightarrow abbaaaA_1AAABB] \Rightarrow abbaaaA_1AAABB_1] \Rightarrow^* abbaaaA_1B_1AAAB] \Rightarrow abbaaaA_1B_1AAAB_1] \Rightarrow^* abbaaaA_1B_1B_1AAA] \Rightarrow abbaaaA_1B_1B_1AAA_1] \Rightarrow^* abbaaaA_1B_1B_1A_1AA] \Rightarrow abbaaaA_1B_1B_1A_1AA_1] \Rightarrow abbaaaA_1B_1B_1A_1A_1A] \Rightarrow abbaaaA_1B_1B_1A_1A_1A_1] \Rightarrow^*$ abbaaaabbaaa

Exercise 1.4.5

Write the grammar, choosing the one with lower power between the regular ones, context free and unrestricted, for the following languages:

1. Language of the algebraic expressions: sums and multiplications in two variables *a* and *b*.
2. $\{a^nb^n \mid n\geq1\} \cup \{a^nb^{2n} \mid n\geq1\}$.
3. {x | the number of *a*’s in x is equal to the number of *b*’s in x};
4. $\{a^{2^i} \mid i\geq0\}$;
5. $\{a^nb^{n^2} \mid n\geq1\}$.

Solution to point 1.

$V_T = \{a, b, +, *, (,)\}$, $V_N = \{S\}$, $P = \{\ S \to a \mid b \mid S + S \mid S * S \mid (S)\ \}$

For instance, a derivation of *a + b*(a)* is:

$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S * S \Rightarrow a + S * (S) \Rightarrow a + b * (S) \Rightarrow a + b * (a)$.

The grammar is context free. The grammar is also the one at minimum power, because the language is not regular.

This could be demonstrated for example using the pumping lemma for context free languages:

x = (a+b)

x = uywzv

y = (a+

w = b

z =)

u = v = ε

$\forall$ i $\geq$ 0, uy^iwz^iv belongs to the language. Indeed b, (a+b), (a+(a+b)), (a+(a+(a+b))), (a+(a+(a+(a+ b)))), ecc. belong to the language.

Solution to point 2.

$V_T = \{a, b\}$, $V_N = \{S_1, S_2, S\}$, $P = \{S \rightarrow S_1 \mid S_2,\ S_1 \rightarrow a\ S_1\ b \mid ab,\ S_2 \rightarrow a\ S_2\ bb \mid abb\}$.

A derivation of *aabbbb* is:

$S \Rightarrow S_2 \Rightarrow a\ S_2\ bb \Rightarrow aabbbb$.

The grammar is context free. The grammar is also the one at minimum power, because it can be proved, using for instance the pumping lemma, that the language is not regular.

Solution to point 3.

It can be demonstrated that no *linear* (i.e., having at most one non terminal in the *right* side of each rule) context free grammar can produce a language of this kind. A context free grammar is however sufficient. The easiest example is shown in the following productions:

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$.

For example, *abbaaaabbb* is obtained by the following derivation:

$S \Rightarrow aSbS \Rightarrow abS \Rightarrow abbSaS \Rightarrow abbaS \Rightarrow abbaaSbS \Rightarrow abbaaaSbSbS \Rightarrow abbaaaaSbSbSbS \Rightarrow abbaaaabSbSbS \Rightarrow abbaaaabbSbS \Rightarrow abbaaaabbbS \Rightarrow abbaaaabbb$.

The correctness of the grammar can be demonstrated in this way. Le L be the language of the exercise, L_b = {y | the number of *a* in y is 1 lower with respect to the number of *b* in y}, L_a = {z | the number of *a* in y is 1 grater with respect to the number of *b* in z}. It is obvious that: if $S \Rightarrow^* x$ then $x \in L$; if $SbS \Rightarrow^* y$ then $y \in L_b$; if $SaS \Rightarrow^* z$ then $z \in L_a$. In order to demonstrate that if $x \in L$ then $S \Rightarrow^* x$, it is better to preceede for induction, with the following three inductive hypothesis:

Given $x \in \{a,b\}^*$, with $1<|x|<n$, then:

i. if $x \in L$ then $S \Rightarrow^* x$;

ii. if $x \in L_b$ then $SbS \Rightarrow^* x$;

iii. if $x \in L_a$ then $SaS \Rightarrow^* x$.

The basic step is obvious (x = ε, x =a, x = b). For the induction step, to demonstrate the (ii) for |x| = n, one just need to observe that if x has a *b* more with respect to the *a*'s, then x is of the kind wbv, with $w, v \in L$: for the inductive hypothesis (i), being $|w| \leq n-1$ and $|v| \leq n-1$, $S \Rightarrow^* w$ and $S \Rightarrow^* v$, and hence $SbS \Rightarrow^* wbv = x$. Similarly for the (iii). For the case (i), observe that if x is of the kind ay, then $y \in L_b$, that is for the (ii), being $|y|= n-1$, $SbS \Rightarrow^* y$, i.e. $S \Rightarrow aSbS \Rightarrow^* ay = x$. The case x = bz is completely symmetrical.

Another grammar for L is for example:

$S \rightarrow aAS \mid bBS \mid \varepsilon$,

$A \rightarrow aAA \mid bS$

$B \rightarrow bBB \mid aS$

where the non-terminal A generates strings in which the number of *b*'s is equal to the number of *a*'s plus one, while B generates strings in which the number of *b*'s is equal to the number of *a*'s plus one.

Solution to point 4.

The required language is not generable by any context free grammar. A non limited grammar must therefore be used.

The building of these grammars is similar to the definition of algorithms (not deterministic) for the single tape TM, with infinite tape in both sides.

As non-terminal alphabet V_N consider the set {A, C, X, Y}. The symbols X and Y are used as end-makers, to indicate the string bounds. C is a symbol used to double the number of A's. The productions are built this way:

$S \rightarrow XAY$: at the beginning there is only one A.

$X \rightarrow XC$: at the right of the end-marker X is possible to generate an arbitrary number of C's.

$CA \rightarrow AAC$: each C can be moved to the right, doubling the A's between X and Y.

$CY \rightarrow Y$: when C meets the right end-marker Y, it can be deleted;

$A \rightarrow a$: the A's can become terminal symbols.

$X \rightarrow \varepsilon$, $Y \rightarrow \varepsilon$: when the generation is over the end-markers can be deleted.

An example of derivation for the string *aaaa* is the following:

$S \Rightarrow XAY \Rightarrow XCAY \Rightarrow XAACY \Rightarrow XAA\ Y \Rightarrow XCAAY \Rightarrow XAACAY \Rightarrow XAAAACY \Rightarrow XAAAAY \Rightarrow XAAAA \Rightarrow XAAAa \Rightarrow XAAaa \Rightarrow XAaaa \Rightarrow Xaaaa \Rightarrow aaaa$.

Solution to point 5.

In order to obtain the required language it is necessary to use a non limited grammar. It is useful to exploit the properties $(n+1)^2 = n^2 + 2n + 1$ and $1^2=1$, to generate a string of the kind $a^{n+1}b^{(n+1)^2}$ from a string of the kind $a^n b^{n^2}$.

Let us define a grammar composed of 9 productions and 6 non-terminal symbol (it is possible to reduce these dimensions).

$S \rightarrow XAYb$: the grammar uses a left end-marker X and a symbol Y that separates the A's from the *b*'s.

$X \rightarrow XC$: at the right of the end-marker X is possible to generate an arbitrary number of C's.

$CA \rightarrow ACB$: this production allows to generate a B for each A, moving C to the right;

$BA \rightarrow AB$ e $BY \rightarrow Ybb$ moving the B's to the right; each B is trasformed in a couple of b at the right of the mark Y.

The result of the application, of the last two productions is the generation of a number of *b's* at the right of Y which is equal to the double of the number n of A's at left to of Y.

The result of the application of the last two productions is the generation of a number of *b's* at the right of Y equal to the double of the number n of A's at left to of Y.

CY $\rightarrow$ AYb: adds the 2n+1-th *b* at the right of Y and the n+1-th A at the left of Y, deleting at the same time the C. The result of all the transformations seen as far, applied until is possible, is to bring from n to n+1 the number of A's and from n^2 to $n^2 + 2n + 1 = (n+1)^2$ the number of b's.

X $\rightarrow \varepsilon$, A $\rightarrow$ a, Y $\rightarrow \varepsilon$: these productions are used to stop the generation of A's and of *b's,* to transform the A's in *a*'s and to eliminate the marks X and Y.

An example of derivation for the string *aabbbb* is the following:
S $\Rightarrow$ XCAYb $\Rightarrow$ XACBYb $\Rightarrow$ XACYbbb $\Rightarrow$ XAAYbbbb $\Rightarrow$ AAYbbbb $\Rightarrow$ aAYbbbb $\Rightarrow$ aaYbbbb $\Rightarrow$ aabbbb.

Exercise 1.4.6

Build a grammar that generates the language L consisting of all the non-empty strings of characters a, b, c such that the number of a is equals to the sum of the number of b's and c's.

It is preferable that the grammar belongs to the minimum generative power class.

Also say what is the class of automata at minimum power that contains an automaton that recognizes L.

Solution

S $\rightarrow$ KX | aY

X $\rightarrow$ a | aS | Sa

Y $\rightarrow$ K | KS | SK

K $\rightarrow$ b | c

The nonterminal X generates strings in which the number of *a*'s is equal to the number of *b*'s plus the number of *c*'s plus one. The nonterminal Y generates strings in which the number of *b*'s plus the number of *c*'s is one greater with respect to the number of *a*'s.

L can be recognized by a deterministic pushdown automaton, but not by finite state automata.

A DPDA which recognizes the language can be easily obtained from the DPDA of Exercise 1.2.3, modifying it in a way that the b's and c's are treated alike.

Exercise 1.4.7

Consider a definition of unrestricted grammars in which, mantaining unchanged the rest, the productions are of the kind

(1) $\alpha N \beta \rightarrow \gamma$ with $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$, $N \in V_N$

(i.e. the left side contains at least one non terminal symbol, and possibly terminal symbols) while in the common definition the productions are of the kind

(2) $\alpha \rightarrow \beta$ with $\alpha \in V_N^+$, $\beta \in (V_N \cup V_T)^*$

(i.e. the left side is a string composed exclusively by non terminal symbols).

Relate the generative ability of the two kind of grammars (1) and (2).

Define a grammar of kind (1) and one of kind (2) for the language

$$L = \{ a^n b^n c^n \mid n \geq 1 \}.$$

Solution

The grammars of kind (1) are surely as powerful as the ones of kind (2) at least: indie grammars of kind (2) are a particular case of the ones of kind (1). The only issue open remains if the grammars of kind (1) are *more* powerful than the ones of kind (2). To prove that it is not true we will show that the set of the languages generated by the grammars of kind (1) is a subset of the ones generated by the grammars of kind (2). The two sets therefore coincide, and the two kinds of grammars have the same generative power.

For this purpose we provide a systematic process to obtain, given a grammar of kind (1), $G_1=<V_{N1}, V_{T1}, S_1, P_1>$, a grammar of kind (2), $G_2=<V_{N2}, V_{T2}, S_2, P_2>$, equivalent, i.e. that generates tha same language.

The alphabet of G_2 is defined as follow:

$$V_{T2} = V_{T1};$$

$$V_{N2} = V_{N1} \cup \{N_a \mid a \in V_{T2}\};$$

N_a is a non terminal that represents the terminal *a* in G_2.

Moreover we have $S_2 = S_1$.

Each derivation in G_1 is "simulated" by a corresponding derivation in G_2 in which is generated a string with the same structure of the one generated by G_1, but with non terminals of convenience N_a in place of each terminal *a* that appears in the string generated by G_1. The terminal symbols are derivated from the non terminal of kind N_a by means of productions of the type "$N_a \rightarrow a$".

The productions of G_2 are hence defined as follow.

Let us define, for each $\alpha \in (V_{T1} \cup V_{N1})^*$, the string $\underline{\alpha} \in V_{N2}^*$ obtained from α by replacing all the terminal $a \in V_{T1}$ with the corresponding new non terminal $N_a \in V_{N2}$ and leaving unchanged the rest. Then we have:

$$P_2 = \{ \underline{\alpha} \rightarrow \underline{\beta} \mid (\alpha \rightarrow \beta) \in P_1 \} \cup \{ N_a \rightarrow a \mid a \in V_{T2} \}$$

It is now easy to show that if G_1 is a generic grammar of kind (1) and G_2 is the corresponding grammar of kind (2) obtained with the rules outlined above, we have $L(G_1) = L(G_2)$, that is (a) $L(G_1) \subseteq L(G_2)$ and (b) $L(G_2) \subseteq L(G_1)$. The proof of these two fact is just outlined in the following.

(a) For all $\alpha \in L(G_1)$, is possible to prove by induction on the lenght of the derivation of α that $S_2 \Rightarrow^* \underline{\alpha}$; then we have $\underline{\alpha} \Rightarrow^* \alpha$ by a sequence of applications of productions of the kind $N_a \rightarrow a$.

(b) It is possible to demonstrate that for each $\delta \in L(G_2)$ exists a derivation of δ in which are applied only productions of the kind $\underline{\alpha} \rightarrow \underline{\beta}$ at first and then only productions of the kind $N_a \rightarrow a$; in other words we have $S_2 \Rightarrow^* \underline{\delta} \Rightarrow^* \delta$. It is easy to show, by induction, that with the same number of derivation steps that gives $S_2 \Rightarrow^* \underline{\delta}$ in G_2, we have, in G_1, $S_1 \Rightarrow^* \delta$.

Let us define the two grammars of kind (1) and (2) for the language $\{ a^n b^n c^n \mid n \geq 1 \}$ according to the above mentioned criterions.

Grammar G_1 of kind (1)

$S \rightarrow aSBC \mid aBC$	$CB \rightarrow BC$	$aB \rightarrow ab$
$bB \rightarrow bb$	$bC \rightarrow bc$	$cC \rightarrow cc$

Derivation for the string $a^3b^3c^3$ (in every string we underline the part that is expanded in the next derivation).

$S \Rightarrow a\underline{S}BC \Rightarrow a^2\underline{S}(BC)^2 \Rightarrow a^3B\underline{CB}CBC \Rightarrow a^3BBC\underline{CB}C \Rightarrow a^3BB\underline{CB}CC \Rightarrow aa\underline{aB}BBCCC \Rightarrow aaa\underline{bB}BCCC \Rightarrow a^3b\underline{bB}CCC \Rightarrow a^3bb\underline{bC}CC \Rightarrow a^3b^3\underline{cC}C \Rightarrow a^3b^3c\underline{cC} \Rightarrow a^3b^3c^3$.

Corresponding grammar G_2 of kind (2) for the same language.

$S \rightarrow N_aSBC \mid N_aBC$

$CB \rightarrow BC$

$N_aB \rightarrow N_aN_b$

$N_bB \rightarrow N_bN_b$

$N_bC \rightarrow N_bN_c$

$N_cC \rightarrow N_cN_c$

$N_a \rightarrow a$

$N_b \rightarrow b$

$N_c \rightarrow c$

Derivation for the same string $a^3b^3c^3$ in G_2.

$S \Rightarrow N_a\underline{S}BC \Rightarrow {N_a}^2\underline{S}(BC)^2 \Rightarrow {N_a}^3B\underline{CB}CBC \Rightarrow {N_a}^3BBC\underline{CB}C \Rightarrow {N_a}^3BB\underline{CB}CC \Rightarrow N_aN_a\underline{N_aB}BBCCC \Rightarrow {N_a}^3\underline{N_bB}BCCC \Rightarrow {N_a}^3N_b\underline{N_bB}CCC \Rightarrow {N_a}^3{N_b}^2\underline{N_bC}CC \Rightarrow {N_a}^3{N_b}^3\underline{N_cC}C \Rightarrow {N_a}^3{N_b}^3N_c\underline{N_cC} \Rightarrow {N_a}^3{N_b}^3{N_c}^3 \Rightarrow^* a^3b^3c^3$.

Exercise 1.4.8

Consider the following language:

$L = \{a^n b^m, n>0, m = f(n)\}$.

1. Give an example of f(n) for which L is generable by a regular grammar.
2. Give an example of f(n) for which the grammar at minimum power that can generate L is context free.
3. Give an example of f(n) for which the grammar at minimum power that can generate L is unrestricted.

Solution to point 1.

If we consider f(n)=k, where k is a constant not negative integer, the considered language would be $L = \{a^n b^k, n>0, k\geq 0\}$, that is generable by a regular grammar.

Solution to point 2.

Consider f(n)=kn, where k is a positive integer constant or f(n)=n/k, in this case L is generable by a context free grammar.

Solution to point 3.

If we consider $f(n)=n^k$, where k is an integer constant > 1, the grammar at minimum power that can generate L is a unrestricted.

Exercise 1.4.9

The operation merge(L_1,L_2) is defined for the languages L1, L2 as follow:

merge(L_1,L_2) = $\{z \mid z = x_1.y_1.x_2.y_2. \dots x_n y_n$, con $x_1. x_2. \dots x_n = x \in L_1, y_1.y_2. \dots y_n = y \in L_2\}$

Each string x_i y_j, can be empty.

For example, if $L_1 = \{a^n b^n \mid n \geq 1\}$, $L_2 = \{c^n d^n \mid n \geq 1\}$, merge($L_1$,$L_2$) = {acbd, abcd, acacddbb, ccddaabb, …}; the string bacd does not belong to merge(L_1,L_2).

1. Discusses briefly how is it possible, given two grammars G_1 and G_2, that generate L1 and L2 respectively, to build one that generates merge (L1, L2). For simplicity one can assume that the alphabets of L1 and L2 are disjoint.
2. Provide a grammar that generates merge($\{a^n b^n\}$, $\{c^n d^n\}$), observing that is not always true that the grammar generates merge (L1, L2) belongs to the same category of G_1 and G_2.

Solution to point 1.

Let *for example* {a,b} be the alphabet of L_1 and {c, d} the alphabet of L_2. Add to their alphabets the sets {A,B} and {C,D} respectively. For a generic string x of {a,b}* let us define X as the string obtained from x by replacing lower cases with upper cases. Similarly for y x of {c,d}*.

Given G_1 and G_2 let us build a grammar that generates all the strings of the kind XY\$, being \$ a new non terminal.

Then, add the rules:

$AC \rightarrow CA$, $BC \rightarrow CB$, $AD \rightarrow DA$, $BD \rightarrow DB$,

that allow to Exchange characters of L_1 with characters of L_2 without modify their relative order.

Finally add the rules $A\$ \rightarrow \$a$, $C\$ \rightarrow \$c$, …, $\$ \rightarrow \varepsilon$,

To transform the strings in sequenze of terminal characters.

Solution to point 2.

A grammar G that generates merge(L_1,L_2), obtained according to the methos proposed at point one, is the following:

$G \rightarrow G_1G_2\$$

$G_1 \rightarrow AG_1B \mid AB$

$G_2 \rightarrow CG_2D \mid CD$

$AC \to CA$

$BC \to CB$

$AD \to DA$

$BD \to DB$

$A\$ \to \$a$

$B\$ \to \$b$

$C\$ \to \$c$

$D\$ \to \$d$

$\$ \to \varepsilon$,

Exercise 1.4.10

Write the the minimum power grammar that generates each of the following languages:

1. $L_1 = \{a^m b^n c^k \mid m+k=n, m\geq 0, k\geq 0, n\geq 0\}$
2. $L_2 = \{a^m b^n c^k \mid m+k=n, m\geq 0, k\geq 0, n\geq 1\}$
3. $L_3 = \{a^m b^n c^n a^k \mid m+k=n, m\geq 0, k\geq 0, n\geq 0\}$

Solution to point 1:

A minimum power grammar that generates L_1 is:

$S \to AC$

$A \to aAb \mid \varepsilon$

$C \to bCc \mid \varepsilon$

Solution to point 2:

A minimum power grammar that generates L_2 is:

$S \to aAbC \mid AbCc$

$A \to aAb \mid \varepsilon$

$C \to bCc \mid \varepsilon$

Solution to point 3:

A minimum power grammar that generates L_3 is:

$S \to IF \mid \varepsilon$

$I \to aIX \mid L$

$F \to XFa \mid R$

$X \to BC$

$CB \to BC$

$LB \to bL$

$CR \to Rc$

$LR \to \varepsilon$

Exercise 1.4.11

Say, briefly explaining the reasons, whether exists a regular grammar that generates the complement of the language $L = \{a^n b^{n^2} | n >= 1\}$, given in Exercise 1.3.9.

Soluzione

There is not a regular grammar that generates the complement of L.

L is not recognizable by a finite state automaton (the recognition of L requires infinite memory, this statement can also be proved rigorously by using the Pumping Lemma). Since the languages generated by regular grammars coincide with those recognized by finite automata, and these are closed under the complement, if there is such a grammar, L would be recognized by a FSA.

Exercise 1.4.12

Say, justifying the answer, which of the following classes of languages are closed under the '*' operator.

a) regular languages
b) Context-free languages, wich are generated by context-free grammars.
c) Languages generated by any grammar.

Remind the definition of the '*' operator:

$$L^* = L^0 \cup L \cup L^2 \cup L^3 \ldots = \cup_{i=0}^{\infty} L^i = \varepsilon \cup L \cup L.L \cup \ldots.$$

Solution

The three classes are all closed unter the '*' operator.

With respect to the class b) and c), consider any grammar G, whose axiom is S. Add to his vocabulary the nonterminal symbol S_1 (It should not be in the original vocabulary) and rules:

$S_1 \to SS_1 | \varepsilon$.

Take S_1 as the axiom of the enriched grammar. It generates ε, ie, L^0, L (applying the derivation $S_1 \Rightarrow SS_1 \Rightarrow S \Rightarrow$ .... followed by any derivation of G, L^2 (applying the derivation $S_1 \Rightarrow SS_1 \Rightarrow SSS_1 \Rightarrow SS$.... followed by any two derivations of G) and so on.

If G is context-free, this property is also true for the G_1 grammar just obtained. This shows that b) and c) are closed under the '*' operator.

A similar procedure also applies for regular grammars. Let G be a regular grammar. Without altering the nonterminal vocabulary, add for each production of type $A \to a$, a new production $A \to aS$, S being the axiom; add also the production $S \to \varepsilon$ (it is clear that in order to generate the null string, a general grammar must admit any productions of type $B \to \varepsilon$; the definition of regular grammars must conventionally allows these rules, otherwise the property of closure is not valid for this Class of languages).

1.5 Non-deterministic models

Exercise 1.5.1

Define the nondeterministic finite automata that recognize the following languages.

i. Strings $x \in \{0, 1\}^*$ that contain a pair of 0 separated by a string of lenght multiple of 4;

ii. Strings $x \in \{0, 1\}^*$ such that the fifth simbol from right is 1;

iii. Strings $x \in \{a, b, c\}^*$ which have the same value if evaluated from left to right or from right to left with respect to operation (non-associative) shown in **Table 1-2** (e.g., (ab) c = c and a (bc) = a).

	a	b	c
a	a	a	c
b	c	a	b
c	b	c	a

Table 1-2 Definition of an operation on strings.

Solution to point i.

Figure 1-84 shows a NFA that recognizes the language. Note that among the multiple of 4 also includes zero. However it is easy to modify the automaton to accept only strings in which there is a pair of 0 separated by a non-empty string of length multiple of 4.

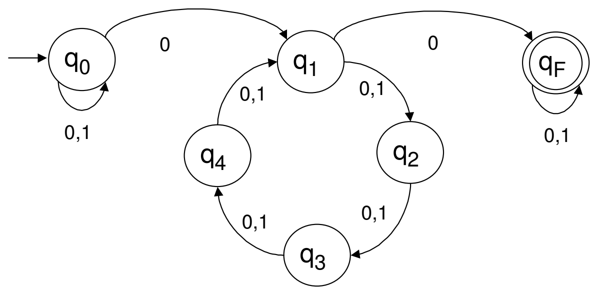


Figure 1-84 NFA that reconigze the language of the Exercise 1.5.1.i.

Solution to point ii.

Figure 1-85 shows a NFA that recognize the language.

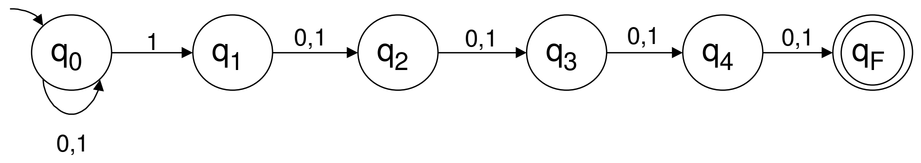


Figure 1-85 NFA that recognizes the language in Exercise 1.5.1.ii.

Solution to point iii.

The automaton is defined by the 5-tuple NFA = <Q, I, δ, q_o, F>, where:

$I = \{a, b, c\}$

$Q = \{q_o\} \cup \{[s, i, d] \mid s, i \in \{a, b, c\}, d \in \{a, b, c, 1\}\}$

Each state, apart from the initial state q_o, is composed by the three s, i, d, with the following meaning:

s is the value of the sub-string scanned so far, computed by associating the operator from left to right;

i is the assumption on the total value of the string (ie the one unread linked with the one already read) computed by combining the operator from right to left; this assumption is made at the beginning of the scan, when the first chatacter is read, and remains the same for all the computation;

d is the value, computed by combining the operator from right to left, that the string still to be scanned need to have so that the total value of the string (Both the already read and unread sub-strings), calculated by combining the operator from right to left , is equal to i.

When a character is read, the value of *d* in the next state is obtained by "dividing" the old value of *d* by the value just read. If the character just read coincides with the value of *d*, a nondeterministic move is to stop, accepting if $s = i$ and setting the *d* of the next state to 1.

$F = \{ [s, i, 1] \mid s = i \} \cup \{q_o\}$

for all $y, i \in I$, $\{ [y, i, d] \mid yd = i \} \subseteq \delta(q_o, y)$, $[y, y, 1] \in \delta(q_o, y)$;

for all $d, y, i \in I$, $[sd, i, 1] \in \delta([s, i, d], d)$, $\{ [sy, i, x] \mid yx = d \} \subseteq \delta([s, i, d], y)$;

for all $s, i, y \in I$, $\delta([s, i, 1], y) = \varnothing$.

Figure 1-58 shows a fragment of the diagram fo the transition function that verifies that abbb$\notin$ L(NFA), while a, ab, abb $\in$ L(NFA).

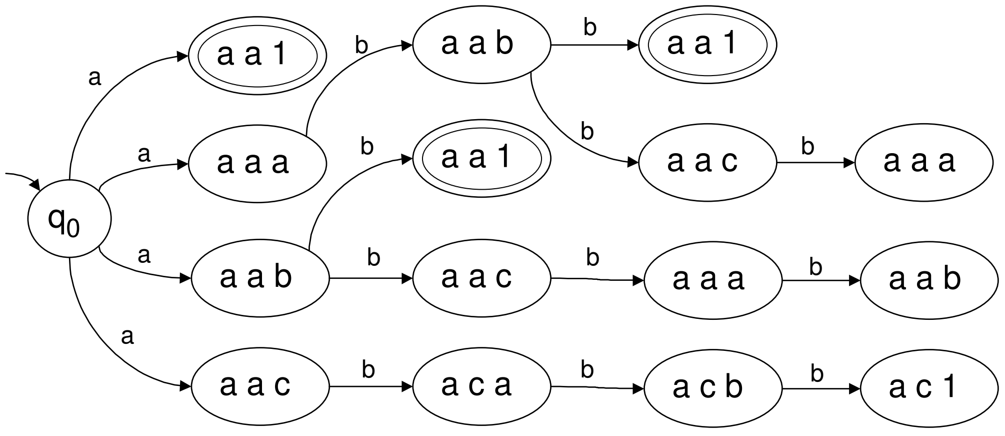


Figure 1-86 A fragment of the NFA that recognizes the language of Exercise 1.5.1.iii.

Exercise 1.5.2

Prove that the class of languages accepted by a finite automaton is closed under the following operations:

Dimostrare che la classe dei linguaggi accettati da un automa finito è chiusa rispetto alle operazioni di

1. union,
2. concatenation,
3. '*', repeated one or more times.

Solution to point 1.

Given two languages L_1 and L_2, there are two finite automata (FA_1 and FA_2), which accept them. From these automata it is possible to construct a nondeterministic finite automaton that accepts $L_1 \cup L_2$. For the well-known equivalence between deterministic and nondeterministic finite automata, it is also possible to define a deterministic finite automaton accepting the same language. Let

$FA_1 = (Q_1, I_1, q_{o1}, \delta_1, F_1)$ e $FA_2 = (Q_2, I_2, q_{o2}, \delta_2, F_2)$;

Assuming, without losing generality, that $Q_1 \cap Q_2 = \emptyset$, and $q_o \notin Q_1 \cup Q_2$

The automaton that accepts FA L1 ∪ L2 is defined as follows:

$FA = (Q, I, q_o, \delta, F\}$

where:
$Q = Q_1 \cup Q_2 \cup \{q_o\}$;
$I = I_1 \cup I_2$;
$F \supseteq F_1 \cup F_2$ and
$q_o \in F$ if $q_{o1} \in F_1$ or $q_{o2} \in F_2$, and no other state belongs to F.

The transition function δ is defined according to the following terms, as shown in Figure 1-59.

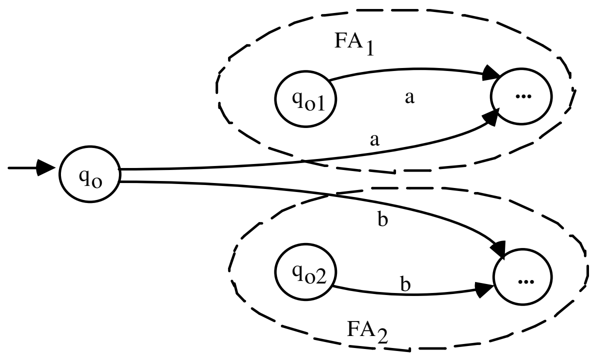


Figure 1-87 A NFA that recognizes the language L(FA1) ∪ L(FA2).

$\forall a \in I_1 \cup I_2$, $\delta_1(q_{o1}, a) \in \delta(q_o, a)$ e $\delta_2(q_{o2}, a) \in \delta(q_o, a)$

$\forall q, q' \in Q_1 \cup Q_2$, $\forall a \in I_1 \cup I_2$, se $\delta_1(q, a)=q'$ then $q' \in \delta(q, a)$

se $\delta_2(q, a)=q'$ then $q' \in \delta(q, a)$

Solution to point 2.

For the known equivalence between finite automata and regular grammars, the language L_1 is generated by a regular grammar RG1 and L2 by RG_2. Let

$RG_1 = <V_{T1}, V_{N1}, P_1, S_1>$ e $RG_2 = <V_{T2}, V_{N2}, P_2, S_2>$;

it is possible to suppose, without losing generality, that: $V_{N1} \cap V_{N2} = \emptyset$ e $S \notin V_N \cup V_N$.

The regular grammar that generates the language $L_1 \cdot L_2$ is the following:

$RG = <V_T, V_N, P, S>$

where $V_T=V_{T1} \cup V_{T2}$; $V_N=\{S\} \cup V_{N1} \cup V_{N2}$, and the set of the productions P is defined as follows:

$P \supseteq P_2$;

$(S \rightarrow S_1) \in P$;

$\forall A \in V_{N1}$, $\forall a \in V_{T1}$, if $(A \rightarrow a) \in P_1$ then $(A \rightarrow a\ S_2) \in P$;

if $(A \rightarrow \varepsilon) \in P_1$ then $(A \rightarrow S_2) \in P$;

if $(A \rightarrow a\ B) \in P_1$ then $(A \rightarrow a\ B) \in P$.

Solution to point 3.

The regular grammar that generates L1*, is RG = <V_{T1}, V_{N1}, P, S_1>, where the set P of productions is defined by the following terms.

$P \supseteq P_1$;

$(S_1 \rightarrow \varepsilon) \in P$;

$\forall A \in V_{N1}, \forall a \in V_{T1}$, if $(A \rightarrow \varepsilon) \in P_1$ then $(A \rightarrow S_1) \in P$;

if $(A \rightarrow a) \in P_1$ then $(A \rightarrow a\, S_1) \in P$.

Exercise 1.5.3

Consider the following language:

$L = \{ab^{n_1}ab^{n_2}...ab^{n_k} | \; k \geq 2, n_i > 0$ per $i = 1, ..., k, \exists \; i, j, 1 \leq i < j \leq k, n_i = n_j\}$

Build a grammar, preferably of the minimum power, that generates L. Find out which is the minimum class of automata that recognize L.

Soluzione

$S \rightarrow ABA \mid AB \mid B \mid BA$

$A \rightarrow aDA \mid aD$

$D \rightarrow bD \mid b$

$B \rightarrow aH$

$H \rightarrow bHb \mid bCb$

$C \rightarrow Aa \mid a$

L can be recognized by a non-deterministic Push-down Automata, but a deterministic PDA cannot recognize it.

A ND PDA can read a generic sequence like $ab^{n_1}ab^{n_2}...ab^{n_h}$ checking if every a is followed by a positive number of b and then, non deterministically, the automata can decide to count $b^{n_{h+1}}$ pushing them into the stack. In the same way, non deterministically, the automata can decide which one of the following $b^{n_{h+1}}$ sequences needs to be compared with $b^{n_{h+1}}$. The automata can do this popping the characters from the stack. Intuitively, this operation cannot be done by deterministic PDA, but a formal proof is complex. This operation can be performed deterministically using an automata belonging to a more powerful class than PDAs.

Exercise 1.5.4

Build a deterministic FSA equivalent to the non-deterministic FSA in Figura 1-88.

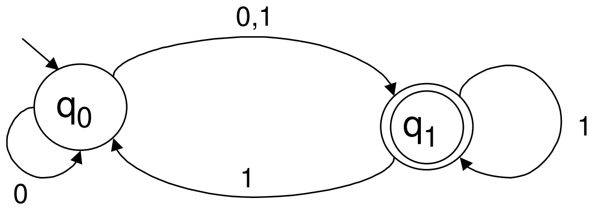


Figura 1-88 A non-deterministic FSA.

Solution

A deterministic FSA M' can be built starting from the deterministic FSA M = (Q, I, δ, q_0, F) in this way:

M' = (2^Q, I, δ', [q_0], {p ∈ 2^Q | ∃q q ∈ p ∧ q ∈ F}) = ($2^{\{q_0, q_1\}}$, I, δ', [q_0], {[q_1], [q_0, q_1]}). M' simulates M keeping track of all the states in which M can simultaneously be during the reading of the input. The building of δ' starts from the initial state q_0 checking in which set of states M can be after every transition. It can be useful to use **Table 1-3**, representing δ.

Stato corrente	Input	Stato prossimo
q_0	0	q_0
q_0	0	q_1
q_0	1	q_1
q_1	0	⊥
q_1	1	q_1
q_1	1	q_0

Table 1-3 Representtion of the transition function δ.

In the example, δ(q_0, 0) = {q_0, q_1}, so δ'([q_0],0) = [q_0, q_1]; δ([q_1], 1) = {q_0, q_1} and then δ'([q_1],1) = [q_0, q_1]. The transitions from q_0 with input 1 and those from q_1 with input 0 are deterministic, so, trivially, δ'([q_0],1) = [q_1] and δ'([q_1],0) = ⊥.

Of course, it is necessary to define the transition function even for the new state [q_0, q_1]: since δ(q_0,0) = {q_0, q_1} and δ(q_1,0) = ⊥ then δ'([q_0, q_1],0) = [q_0, q_1]; since δ(q_0,1) = {q_1} and δ(q_1,1) = { q_0, q_1} then δ'([q_0, q_1],1) = [q_0, q_1].

In Figure 1-89 the obtained automata is shown. It is easy to see that the automata recognizes every non-empty string of 0 and 1, but those starting with 10.

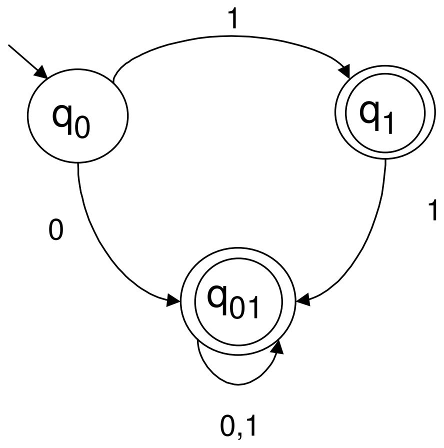


Figure 1-89 A deterministic FSA equivalent to the deterministic FSA in Figura 1-88.

Exercise 1.5.5

Build a NPDA for the language L = $\{ww^R| \ w \in \{a,b\}^+\}$, i.e. the language of palindromes without a center marker. Is it possible to build a DPDA for L?

Solution

The language of palindromes with a center marker $\{wcw^R| \ w \in \{a,b\}^+\}$ is recognizable by a deterministic pushdown automaton, as shown in Exercise 1.2.1, by first storing the w part of the string (i.e. the substring up to the first c) on the stack and then comparing the rest of the input with the rest of the stack. This is possible thanks to the marker c that let decide when it ended the first portion of the input. In the language without center marker it is not possible to decide deterministically when you finished reading the first portion of the string. The automaton cannot know the length of the input string until it has consumed the entire string, and without knowing the length of the string it cannot know which character is the center. A nondeterministic automaton has, however, the following important property: if more moves are possible from the same configuration, the automaton always chooses ("guess") one of the moves that can bring to a state of acceptance, if there is at least one.

In the example, it is possible to build a nondeterministic automaton, equal to the deterministic one of Exercise 1.2.1, with the exception of the move that recognizes the marker c and changes the state from q_0 to q_1. That move is substituted by an ε-move. In this way, while the automaton is in q0 (And this is saving the entry on the stack), it always has the choice between two possible moves: to continue to stack (staying in q0) or begin to start comparing the stack with the input (move to q1). For the property of the nondeterministic automaton, it goes form q0 to q1 only once, exactly at the right time, guessing at each step which is the appropriate move. A graphical representation of the automaton is shown in Figure 1-90.

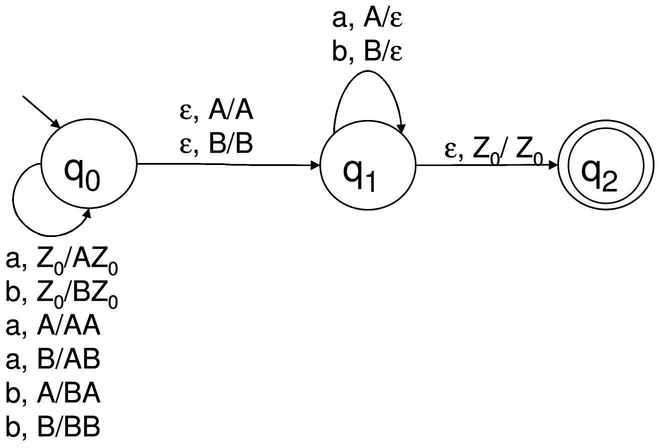


Figure 1-90 An NPDA that recognizes $L = \{ww^R \mid w \in \{a,b\}^+\}$.

Exercise 1.5.6

Build a PDA that recognizes the following language:

$L = \{a^n b^n c^n \mid n \geq 1\}^\wedge$ (The complement operator $^\wedge$ is meant to be applied in relation to the universal language $\{a,b, c\}^*$).

Observe that L can be defined as $L = \varepsilon \cup \{a^* b^* c^*\}^\wedge \cup \{a^n b^m c^k \mid n \neq m \vee m \neq k\} = \{a^* b^* c^*\}^\wedge \cup \{a^n b^m c^* \mid n \neq m\} \cup \{a^* b^n c^m \mid n \neq m\}$. The strings that belongs to the language are:

- the empty string;
- all the strings containing the characters *a*, *b* e *c*, in which all the *a* does not come before all the *b* and all the *b* does not come before all the *c*;
- all the strings containing *a*,*b*, and c in which all the *a* are before the all the *b*, all the *b* are before the all the *c* and the number of *b* is different than the number of *a* or the number of *c*.

Fromally, L can be defined as $L = L_1 \cup L_2 \cup L_3$ where

$L_1 = \varepsilon \cup \{a^* b^* c^*\}^\wedge$

$L_2 = \{a^n b^m c^* \mid n \neq m\}$

$L_3 = \{a^* b^n c^m \mid n \neq m\}$

L_1 is a regular language and it is recognized by the FSA reported in Figure 1-91. The automaton is easily obtainable from the FSA that recognizes a*b*c*.

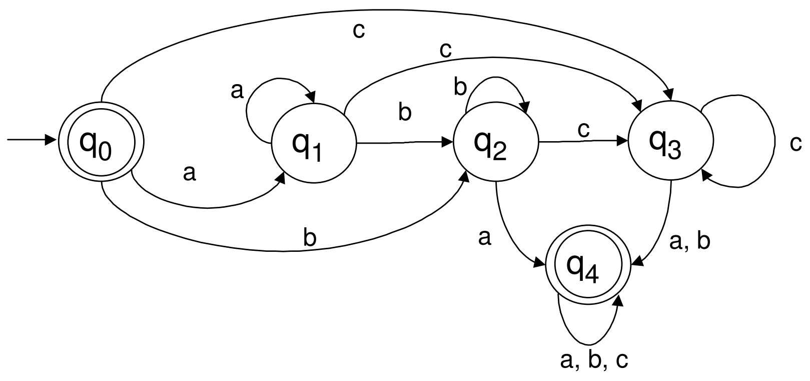


Figure 1-91 A NPDA that recognizes $\varepsilon \cup \{a^* b^* c^*\}^\wedge$.

L_2 is accepted by the PDA reported in Figure 1-92.

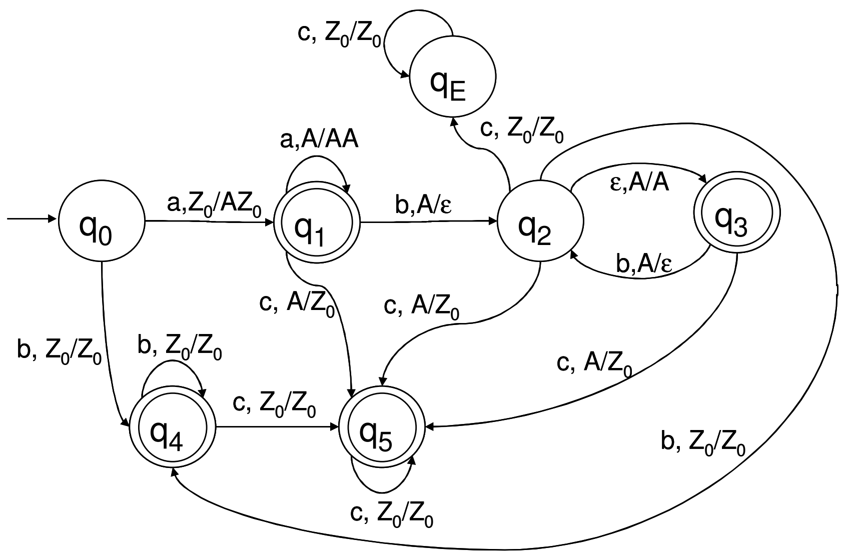


Figure 1-92 A NPDA that recognizes $L_2 = \{a^n b^m c^* | n \neq m\}$.

L_3 is accepted by a similar PDA. L is accepted by the NPDA obtainable with the union of the previous automata.

1.6 Specification of sistems and their properties with logic formulas

Exercise 1.6.1

Let r be a person's income; let $l_1, l_2, ..., l_k$, $l_1<l_2$, $l_2<l_3$, ..., k be costant values denoting different income class; let $a_1, a_2, ... a_k, a_{k+1}$ be k+1 increasing constants denoting the different tax rates.

1. Specify with a first order formula the problem of calculating the function t(r) denoting the tax that a person with income r has to pay, considering the current progressive taxing system: for the first income class you pay the rate a_1, for the second you pay the rate a_2, …, for the income exceeding l_k limit you pay maximum rate a_{k+1}.
2. (*) Solve the problem of the previous point assuming that the number k of income

classes and corresponding rates is variable.

Solution to point 1.

The function t(r) is specified by the following formula (that for shortness isn't written in in a first order formula in a strict sense, but it can immediately be transcribed with a pure first order predicate syntax).

$$\forall r((r \le l_1) \to (t(r) = r.a_1) \wedge$$
$$(l_1 < r \le l_2) \to (t(r) = l_1.a_1 + (r - l_1).a_2) \wedge$$
$$\ldots$$
$$(l_k < r) \to (t(r) = l_1.a_1 + \ldots(l_k - l_{k-1}).a_k + (r - l_k).a_{k+1})$$

Solution to point 2.

n the second case, it's not possible to formalize income classes and tax rates neither as constants nor as variables, because their number depends on k which is variable itself.

However it is possible to formalize that as function of integers, respectively l(i) and a(i), and then to consider their values for the argument values ≤ k.

Moreover, in the first case, is has been used a formula to indicate the repetition. That formula expressed all the possible cases (the ellipsis should be intended only as an abbreviation of the various elements off the formula that being fixed can always be expanded getting to the correct syntax of the first order predicates). When k is variable, there is the need to formalize the sum concept in the first order syntax.

Therefore lets introduce the well known symbol $\sum_{i=h}^{k} f(i)$ as a rewriting of the term Sum_f(h,k) where the function Sum_f satisfies the following axioms:

$\forall$ h, k (k< h) $\to$ Sum_f(h,k) = 0 $\wedge$ (k $\ge$ h $\to$ Sum_f(h,k) = Sum_f(h, k-1) + f(k))

Said that, we can formalize taxation with k tax rates as follows:

$\forall$ r,k

$\big(r \le l(1) \to t(r,k) = r \times a(1)\big) \wedge$

$\big(\exists i(((1<i\le k) \wedge l(i-1)<r\le l(i)) \to t(r,k) = \sum_{h=2}^{i} l(h-1)\times a(h-1) + (r-l(i-1))\times a(i))\big) \wedge$

$\big(r>l(k) \to t(r,k) = \sum_{h=1}^{k} l(h)\times a(h) + (r-l(k))\times a(k+1)\big)$

Exercise 1.6.2

1) Formalize with first order logic formulas the following temporized switch behavior:
   a. If the switch is clicked when the light is off, the light turns on and remains on for Δ time units, after that it turns off automatically.
   b. If the switch is clicked when the light is already on, nothing happens (i.e. the light turns off after the predicted time, as if the switch wasn't pressed again).

Hint: use a “time” variable *t* and predicates like click_switch(t), light_on(t), light_off(t), …

2) Modify the previous formulation to change the behavior of the switch as follows:
   a. If the switch is clicked when the light is off, the light turns on and remains on for

Δ time units, after that, if the switch isn't clicked again, it turns off automatically.

b. If the switch is clicked when the light is already on, it remains on for a time Δ starting from the last time the switch was clicked

Solution to point 1.

Similarly to what happens in the system specified in the **Error! Reference source not found.**, it is possible to notice that, strictly speaking, the problem formulation doesn't exclude the possibility that the light turns on spontaneously without the switch being clicked. In this case however, we assume that this requisite is implicit in the problem formulation.

To formalize the problem, it is possible to use the following predicates:

- light_on(t),
- light_off(t),
- switch_pressed(t),
- turning_on(t),
- turning_off(t)

The switch behavior is then described by the following formulas:

$\forall t\ (\text{light\_on}(t) \leftrightarrow \neg\text{light\_off}(t))$

$\forall t\ (\text{turning\_on}(t) \leftrightarrow \text{light\_off}(t) \wedge \text{switch\_pressed}(t))$

$\forall t\ (\text{turning\_on}(t) \rightarrow (\forall t_1\ (t<t_1<t+\Delta) \rightarrow \text{light\_on}(t_1) \wedge \text{turning\_off}(t+\Delta)))$

$\forall t\ (\text{turning\_off}(t) \rightarrow \text{light\_off}(t))$

$\forall t\ (\text{light\_on}(t) \rightarrow \exists\ t_1\ (t<t_1<t+\Delta) \wedge \text{turning\_off}(t_1) \wedge \exists\ t_2\ (t-\Delta<t_2<t) \wedge (\text{turning\_on}(t_2))$

Notice that it has been adopted the convention that the light is always on in open intervals of time, and off in closed intervals (included the case that is off only for an instant: in fact, it could happen a switch on and a switch off at the same time)

Solution to point 2.

The formulas of the solution to point 1 have to be modified as follows:

$\forall t\ (\text{light\_on}(t) \leftrightarrow \neg\text{light\_off}\ (t))$

$\forall t\ (\text{turning\_on}(t) \leftrightarrow \text{light\_off}\ (t) \wedge \text{switch\_pressed}(t))$

$\forall t\ (\text{turning\_on}(t) \rightarrow (\forall t_1\ (t<t_1<t+\Delta) \rightarrow \text{light\_on}\ (t_1)))$

$\forall t\ (\text{turning\_off}(t) \rightarrow \text{light\_off}\ (t))$

$\forall t\ (\text{light\_on}\ (t) \rightarrow \exists\ t_2\ (t_2<t) \wedge (\text{turning\_on}(t_2) \wedge \forall\ t_1\ (t_2<t_1<t) \rightarrow \text{light\_on}\ (t_3))$

$\forall t\ (\text{light\_on}\ (t) \wedge \text{switch\_pressed}(t) \rightarrow (\forall t_1\ (t<t_1<t+\Delta) \rightarrow \text{light\_on}(t_1)))$

$\forall t\ (\text{turning\_off}(t) \leftrightarrow (\forall t_1\ (t-\Delta<t_1<t) \rightarrow \text{light\_on}(t_1) \wedge \neg\text{switch\_pressed}\ (t_1))$

Exercise 1.6.3

Using suitable <pre-condition, post-condition> pairs, give a *specification* of the programs or program *fragments* implementing the following functions (you do NOT have to *code* them):

1. given three natural numbers, the program returns 1 if one of the three numbers is the product of the other two numbers;
2. given two natural numbers, the program computes the least common multiple of them;

3. given two integer numbers x and y, the program computes the best rounded down integer solution of $\log_x(y)$.

Note: For the sake of simplicity, it is sufficient to specify a program fragment, assuming that inputs and result are memory cells, do not take care about input/output operations. Always call inputs x, y, w and the result z.

Solution to point 1.

Pre: $x \in N \wedge y \in N \wedge w \in N$

Post: $(z = 1 \wedge (w = x*y \vee x = w*y \vee y = x*w)) \vee (z = 0 \wedge w \neq x*y \wedge x \neq w*y \wedge y \neq x*w))$

Solution to point 2.

Pre: $x \in N \wedge y \in N$

Post: $\exists\, k_1 (z = k_1*x) \wedge \exists\, k_2 (z = k_2*y) \wedge \neg\exists\, i_1,i_2 (i_1*x = i_2*y \wedge i_1*x <z)$, con $k_1,k_2,i_1,i_2 \in N$

Solution to point 3.

Pre: $x \in N \wedge y \in N \wedge x > 1$

Post: $x^z = y \vee (x^z < y \wedge x^{z+1} > y)$

Exercise 1.6.4

Specify with a first order formula the language produced by the following grammar:

$S \rightarrow AS \mid A$

$A \rightarrow aAb \mid ab$

Solution

$x \in L_A \Leftrightarrow (x = ab \vee \exists\, y\, (x = ayb \wedge y \in L_A))$

$x \in L \Leftrightarrow (x \in L_A \vee \exists\, w,z\, (x = wz \wedge w \in L_A \wedge z \in L)$

Exercise 1.6.5

A notebook can be ON or OFF. Its battery can be more or less charged. Moreover, the notebook can be PLUGGED or not.

When the notebook is unplugged and is on, the battery discharges.

When the notebook is plugged and is on, the battery wont charge, because we assume all the power is needed to supply the notebook.

When the notebook is plugged and is off, the battery will charge.

When the notebook is used on battery and the battery completely discharges, the notebook automatically turns off.

Assume that the maximum working time on battery is Δ (i.e. Is the time needed to go from max charge to complete discharge). The charging time (from zero to max) with notebook off and plugged is also Δ.

The notebook is initially off, unplugged and the battery is discharged.

Figure 1-93 describes one possible event sequence and the consequent notebook states (assume there weren't events before the origin of time).

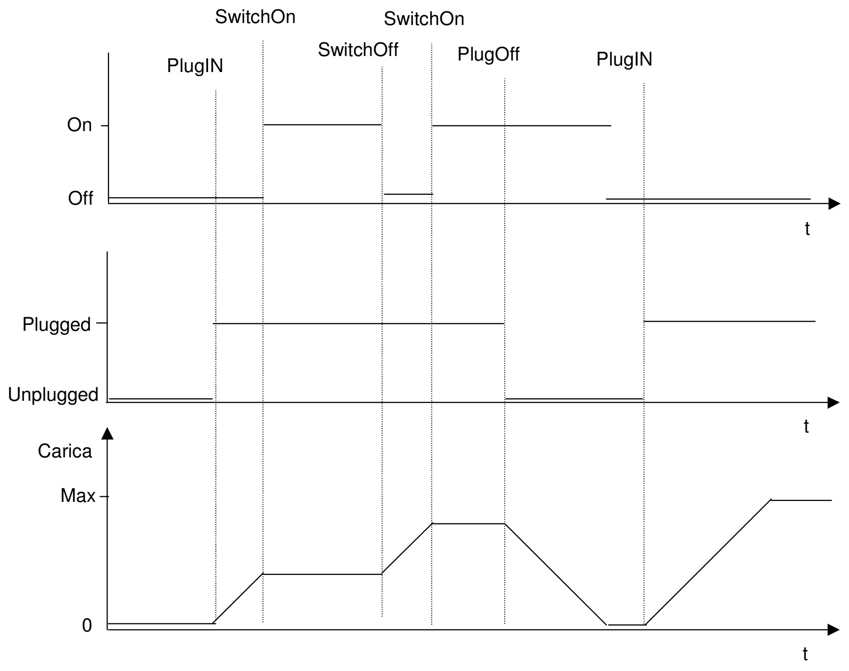


Figure 1-93

Formalize the notebook behavior.

Solution

State predicates:

ON(t), OFF(t), PLUGGED(t), UNPLUGGED(t), CHARGE(tc, t).

CHARGE(tc, t) means that at time *t* the battery was charged for *tc* time units.

Events:

SwitchON(t), SwitchOFF(t), PLUG(t), UNPLUG(t).

Mutual exclusiveness between events:

$\text{SwitchON(t)} \rightarrow \neg\text{SwitchOFF(t)}$

$\text{SwitchOFF(t)} \rightarrow \neg\text{SwitchON(t)}$

$\text{PLUG(t)} \rightarrow \neg\ \text{UNPLUG(t)}$

$\text{UNPLUG(t)} \rightarrow \neg\ \text{PLUG(t)}$

Mutual exclusiveness between states:

$\text{ON(t)} \leftrightarrow \neg\ \text{OFF(t)}$

$\text{PLUGGED(t)} \leftrightarrow \neg\ \text{UNPLUGGED(t)}$

Switch on:

$\text{ON(t)} \leftrightarrow \exists\ t'(t' \leq t \wedge \text{SwitchON(t')} \wedge \forall\ t''\ (t' < t'' \leq t \rightarrow\ \neg\text{SwitchOFF(t'')} \wedge$

$((CHARGE(tc, t'') \wedge tc>0) \vee PLUGGED(t''))))$

Plug in:

$PLUGGED(t) \leftrightarrow \exists t'(t' \leq t \wedge PLUG(t') \wedge \neg\exists t'' (t'< t'' \leq t \wedge UNPLUG(t''))$

Charging doesn't change if the notebook is off and unplugged o on and plugged.

$\exists t' (t' \leq t \wedge CHARGE(tc, t') \wedge \forall t'' (t'\leq t''<t \rightarrow ((OFF(t'') \wedge UNPLUGGED(t''))$
$\vee (PLUGGED(t'') \wedge ON(t'')))) \rightarrow CHARGE(tc, t)$

Charging with notebook off and plugged:

$\exists t' (t' \leq t \wedge CHARGE(tc, t') \wedge \forall t'' (t'\leq t''<t \rightarrow (OFF(t'') \wedge PLUGGED(t''))) \wedge$
$t_k=min(tc+t-t', \Delta)) \rightarrow CHARGE(t_k,t)$

Discharging (notebook on and unplugged):

$\exists t' (t' \leq t \wedge CHARGE(tc, t') \wedge \forall t'' (t'\leq t''<t \rightarrow (ON(t'') \wedge UNPLUGGED(t''))) \wedge tc-$
$(t-t')= t_k \wedge t_k \geq 0) \rightarrow CHARGE(t_k,t)$

Initial state:

$\exists t' (\forall t'' (t''\leq t' \rightarrow (OFF(t'') \wedge UNPLUGGED(t'') \wedge CHARGE(0,t''))))$

Exercise 1.6.6

Consider the behavior of a CD player with only two buttons: PLAY and STOP. Suppose that CD loading and ejecting is manual, so there is no need of an EJECT button.

When the player is in idle state, if it happens that there is a CD in it, it does not move.

When PLAY button is pressed, the device activates the CD rotation. It takes time T to reach the correct speed. When the required speed is reached, the playback starts and it ends when the user pushes the STOP button or when the CD has been fully reproduced.

When STOP button is pressed the device returns to idle state in a fixed time T. During this time, it ignores any command it receives. The device has the same behavior also when a CD has been fully reproduced.

Specify the behavior of the device using first order logic.

Note: the playback can start only when the device has been loaded (there is a CD inside it). Suppose that there is a sensor giving this information. Also suppose that all the CDs have the same length D.

Solution

Predicates indicating the device operating status: Idle(t), Playing(t), Starting_up(t), Shutting_down(t).

Predicate indicating the device content: CD_in(t).

Events: Play(t), Stop(t).

Mutual exclusion among states:

$(Idle(t) \wedge \neg Playing (t) \wedge \neg Starting\_up(t) \wedge \neg Shutting\_down(t)) \vee$
$(\neg Idle(t) \wedge Playing (t) \wedge \neg Starting\_up(t) \wedge \neg Shutting\_down(t)) \vee$

$(\neg \text{Idle}(t) \wedge \neg \text{Playing}(t) \wedge \text{Starting\_up}(t) \wedge \neg \text{Shutting\_down}(t)) \vee$
$(\neg \text{Idle}(t) \wedge \neg \text{Playing}(t) \wedge \neg \text{Starting\_up}(t) \wedge \text{Shutting\_down}(t))$

Mutual exclusion among events (assuming that Play and Stop command are not concurrent):

$\text{Play}(t) \rightarrow \neg \text{Stop}(t)$

$\text{Stop}(t) \rightarrow \neg \text{Play}(t)$

Initial idle condition:

$\forall t' (t' \leq t \rightarrow \neg (\text{CD\_in}(t') \wedge \text{Play}(t')) \rightarrow \text{Idle}(t)$

Definition of Go event:

$\text{Go}(t) \leftrightarrow \text{Idle}(t) \wedge \text{CD\_in}(t) \wedge \text{Play}(t)$

Consequences of Go event:

$(\text{Go}(t) \wedge \forall t'(t<t'\leq t+T \rightarrow \neg \text{Stop}(t'))) \rightarrow$
$\forall t''(t<t''\leq t+T \rightarrow \text{Starting\_up}(t'')) \wedge \text{Start\_play}(t+T)$

$(\text{Go}(t) \wedge \exists K(0<K\leq T \wedge \forall t'(t<t'< t+K \rightarrow \neg \text{Stop}(t')) \wedge \text{Stop}(t+K)) \rightarrow$
$\forall t''(t<t''\leq t+K \rightarrow \text{Starting\_up}(t'')) \wedge \text{Start\_shut\_down}(t+K)$

Consequences of StartPlaying event:

$(\text{StartPlay}(t) \wedge \forall t'(t<t'\leq t+D \rightarrow \neg \text{Stop}(t'))) \rightarrow$

$\forall t''(t<t''\leq t+D \rightarrow \text{Playing}(t'')) \wedge \text{Start\_shut\_down}(t+D)$

$(\text{StartPlay}(t) \wedge \exists K(0<K\leq D \wedge \forall t'(t<t'< t+K \rightarrow \neg \text{Stop}(t')) \wedge \text{Stop}(t+K)) \rightarrow$
$\forall t''(t<t''\leq t+K \rightarrow \text{Playing}(t'')) \wedge \text{Start\_shut\_down}(t+K)$

Consequences of Start_shut_down event:

$\text{Start\_shut\_down}(t) \rightarrow \forall t'(t<t'< t+T \rightarrow \text{Shutting\_down}(t')) \wedge \text{Idle}(t+T)$

Permanency in Idle state (potentially for an undefined amount of time):

$\exists t'(t' \leq t \wedge \text{Idle}(t') \wedge \forall t'' (t'\leq t'' < t \rightarrow \neg \text{Go}(t''))) \rightarrow \text{Idle}(t)$

Exercise 1.6.7

Formalize with the use of first order formulas the behavior of an automatic gate, defined as follows:

- The gate is controlled by a one-button remote control. For simplicity assume that clicking the button is instantaneous.
- When the gate is closed and the remote control button is clicked, the gate starts opening.
- If the button is clicked while the gate is opening or when it is already opened, the signal is ignored.
- After the gate has been opened for X seconds, it starts closing.
- The complete opening and closing movements (i.e. from completely open to completely closed and vice-versa) last Y seconds.

If you find any lack and/or ambiguities in the description, make some reasonable hypothesis explaining the reasons.

You are invited to organize the exposition with the maximum clearness, using a reasonable terminology and -in case of need- accompaning the formulas with a short textual explanation.

Extension 1

If the button is clicked when the gate has been closing for Z seconds, the gate starts opening again and it will completely open after Z seconds.

Extension 2

The gate is equipped with photocells, that signal if the gateway is clear or not.

If the gateway is taken when the gate is closing, it starts opening again. As usual, if the gate was closing since Z seconds it will completely open after Z seconds.

The gate starts closing only if the gateway is clear.

Solution:

Predicates used:

Aperto(t), Chiuso(t), InApertura(t), InChiusura(t) indicano lo stato del cancello.

Opened(t),Closed(t),Opening(t),Closing(t) which represent the state of the gate.

Open (t) states that the opening signal was sent at t.

Mutual exclusiveness between states:

$(\text{Opened}(t) \wedge \neg (\text{Closed}(t) \vee \text{Opening}(t) \vee \text{Closing}(t))) \vee$

$(\text{Closed}(t) \wedge \neg (\text{Opened}(t) \vee \text{Opening}(t) \vee \text{Closing}(t))) \vee$

$(\text{Opening}(t) \wedge \neg (\text{Closed}(t) \vee \text{Opened}(t) \vee \text{Closing}(t))) \vee$

$(\text{Closing}(t) \wedge \neg (\text{Closed}(t) \vee \text{Opened}(t) \vee \text{Opening}(t)))$

Initial state:

$\forall t'(t' \leq t \rightarrow \neg\text{Open}(t')) \rightarrow \text{Closed}(t)$

Compact solution:

$\text{Closed}(t) \wedge \text{Open}(t) \rightarrow (\forall t' \ (t < t' \leq t+Y) \rightarrow \text{Opening}(t')) \wedge (\forall t'' \ (t+Y < t'' \leq t+Y+X) \rightarrow \text{Opened}(t'')) \wedge (\forall t''' \ (t+X+Y < t''' < t+2Y+X) \rightarrow \text{Closing}(t''')) \wedge \text{Closed}(t+2Y+X)$

$\exists t'(t' \leq t \wedge \text{Closed}(t') \wedge \forall t'' \ (t' \leq t'' < t \rightarrow \neg\text{Open}(t''))) \rightarrow \text{Closed}(t)$

Such a solution is not suitable for an incremental addition of extensions.

Alternative solution:

$(\text{Closed}(t) \wedge \text{Open}(t)) \rightarrow (\forall t'(t < t' \leq t+Y) \rightarrow \text{Opening}(t'))$

$(\forall t' \ (t-Y<t' \leq t) \rightarrow \text{Opening}(t')) \rightarrow (\forall t'' \ (t<t'' \leq t+X \rightarrow \text{Opened}(t''))$
$\wedge \text{StartClosing}(t+X))$

$\text{StartClosing}(t) \rightarrow ((\forall t'' \ (t<t''<t+Y) \rightarrow \text{Closing}(t'')) \wedge \text{Closed}(t+Y))$

$\exists t'(t' \leq t \wedge \text{Closed}(t') \wedge \forall t'' \ (t' \leq t'' < t \rightarrow \neg\text{Open}(t''))) \rightarrow \text{Closed}(t)$

Solution taking into account extension 1

There is the need to add the condition regulating the "reopening" if the signal was sent while the gate is closing. In fact, it is no longer true that if the gate has been opened for a period of X, then it will always be closing for a period of Y: it could arrive an opening command during that time. The predicate that regulates the closing has to be rewritten to check the opening commands that could arrive while the gate is closing.

StartClosing(t) → (((∀ t” (t<t”<t+Y) → (¬Open(t”) ∧ Closing(t”))) ∧ Closed(t+Y)) ∨ (∃ K (0<K<Y ∧ (∀t’ (t<t’<t+K) → (¬Open(t’)∧ Closing(t’))) ∧ Open(t+K) ∧ (∀t’” (t+K<t’”≤t+2K) → Opening(t’”)) ∧ (∀ t”” (t+2K< t”’≤t+2K+X) →Opened(t””)) ∧ StartClosing(t+2K+X))

Solution taking into account extension 2

There is the need to redefine Open, which doesn't coincide anymore only with the button click (event Button(t)), but that has to take also int account the change of the state of the gateway from clear to taken.

Open(t) ↔ Button(t) ∨ ∃ t’,t’” ((t’ < t < t’”) → (∀ t” (t’<t”<t) → Clear(t”)) ∧ (∀ t”” (t <t””<t’”) → ¬Clear(t””)))

The gate starts closing only if the gateway is clear: the consequences of StartClosing(t) have therefore to be redefined.

StartClosing(t) ∧ Clear(t) → ... (The consequent of this forumula is the same reported for StartClosing(t) in the solution to the extension 1)

StartClosing(t) ∧ ¬Clear(t) ↔ BeginWaitToClose(t)

(∃ t’ (t’≤t ∧ BeginWaitToClose (t’) ∧ (∀ t” (t’≤t”≤t) → ¬Clear(t”)))) → WaitToClose(t)

Note: WaitToClose(t) is a gate state,that has to be in mutual exclusion with the other possible states (see above).

(WaitToClose(t) ∧ ∃ t’,t’” ((t’ < t < t’”) → (∀ t” (t’<t”<t) → ¬Clear(t”)) ∧ (∀ t”” (t <t””<t’”) → Clear(t””)))) → ... (The consequent of this forumula is the same reported for StartClosing(t) in the solution to the extension 1)

Exercise 1.6.8

Formalize through a first-order logic formula the magnetic hysteresis phenomenon as described in schematic form in the traditional diagram in Figure 1-94.

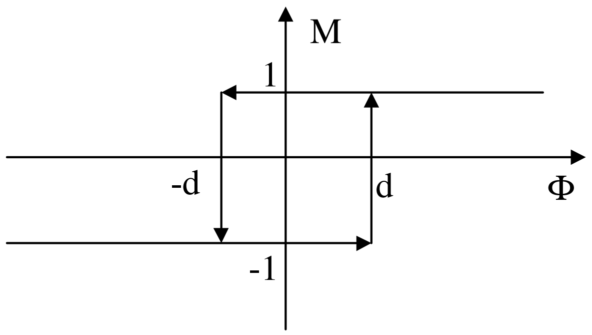


Figure 1-94 The magnetic hysteresis phenomenon

For simplicity we neglect the transition where the magnetization starts within the "rectangle of hysteresis", so we can assume that at any time the magnetization is in one of two stable states (here conventionally indicated with +1 and -1).

Solution

The level of magnetization is described as a time-dependent variable M(t) which can take the values -1 and +1; similarly the magnetizing flux is indicated by Φ(t) which can assume all values on the real axis. Now the phenomenon is described by a formula that expresses the fact that at a generic time t, M (t) = 1 if and only if Φ has previously assumed a value $\geq$d and never under -d; the same behavior happens symmetrically when M(t)=-1.

$M(t) = 1 \leftrightarrow \exists\, t_1\, [(t_1 \leq t) \wedge (\Phi(t_1) \geq d \wedge \forall t_2\, (t_1 \leq t_2 \leq t \rightarrow \neg\, (\Phi(t_2) \leq -d)))]$

$M(t) = -1 \leftrightarrow \exists\, t_1\, [(t_1 \leq t) \wedge (\Phi(t_1) \leq -d \wedge \forall t_2\, (t_1 \leq t_2 \leq t \rightarrow \neg\, (\Phi(t_2) \geq d)))]$

<u>Alternative solutions (it is reported only the first condition being the other symmetrical):</u>

$M(t) = 1 \leftrightarrow \exists\, t_1\, [(t_1 \leq t) \wedge (\Phi(t_1) \geq d \wedge \forall t_2\, (t_1 \leq t_2 \leq t \rightarrow \Phi(t_2) > -d))]$

…

or

$M(t) = 1 \leftrightarrow \exists\, t_1\, [(t_1 \leq t) \wedge (\Phi(t_1) \geq d \wedge \neg\exists t_2\, (t_1 \leq t_2 \leq t \wedge \Phi(t_2) \leq -d))]$

…

A possible alternative solution based on FSA

There are 4 states:

- $q_{Ml\Phi l}$, corresponding to $\Phi < -d$ (and so to $M = -1$);
- $q_{Ml\Phi m}$, corresponding to an average Φ, so $-d < \Phi < d$, and $M = -1$;
- $q_{Mh\Phi h}$, corresponding to $\Phi > d$ (and so to $M = +1$);
- $q_{Mh\Phi m}$, corresponding to an average Φ, so $-d < \Phi < d$, and $M = +1$.

There are 4 events:

- lm, correspondening to the transition of Φ from a value lower than -d to a value greater than -d;
- mh, correspondening to the transition of Φ from an average value (I.e. $-d < \Phi < d$) to a value greater then d.
- hm, corresponding to the transitino of Φ from a value greater than d to an average value (I.e. $-d < \Phi < d$);
- ml, corresponding to the transitino of Φ from an average value (I.e. $-d < \Phi < d$) to a value lower than -d.

The automaton that describes the system is reported in Figure 1-95.

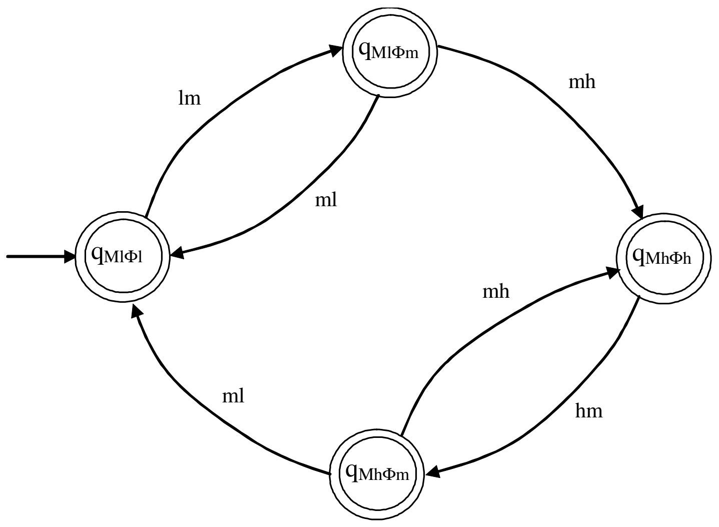


Figure 1-95 The automaton which represents the phenomenon of magnetic hysterisis.

Of course, to be precise there is the need need to define the events in a proper manner. For example, you can use the following predicates:

$lm(t) \leftarrow \exists\ t_m, t_l\ ((t_m>t \wedge t_l<t) \rightarrow (\Phi(t_l) <-k \wedge \Phi(t_m) >-k \wedge \Phi(t_m) <k))$

$mh(t) \leftarrow \exists\ t_m, t_h\ ((t_m<t \wedge t_h>t) \rightarrow (\Phi(t_m) >-k \wedge \Phi(t_m) <k \wedge \Phi(t_h) >k))$

...

Exercise 1.6.9

Specify (don't *code*!) with an appropriate pair of <pre-, post-condition>, a program or a fragment of program that, given a prime *p*, it sums 1 to it and divides by 2, and then divides by 2 as long as possible, indicating with *z* the final result -integer- obtained this way (e.g. For p=19 the program should return z=5)

Note: for simplicity, it is sufficient to specify a fragment of program, assuming that the imput *p* and the output *z* are memoty cells, ignoring the input/output operations.

Solution

Using Hoare's notation the required specification can be expressed with the following <pre-, post-condition> pair:

Pre: $\neg\exists y, u\ (y > 1 \wedge u > 1 \wedge p = y{*}u)$ {p is prime}

Post $\exists k\ (\ k \geq 1 \wedge z{*}2^k = p + 1 \wedge \neg\exists h\ ((h > k)) \wedge z{*}2^h = p + 1))$

{z multiplied by a natural power of 2 has to give p + 1, and there should not be any greater power of two which divides *p+1*}

Exercise 1.6.10

Formalize the following definition of a personal computer boot procedure:

When the power switch is pressed, the system automatically load the OS and the application A (e.g. A word processor or a mail client), unless within 3 seconds from the start the user press the SHIFT key, in that case the system only loads the OS. If the SHIFT key is not pressed, the boot completes within 15 seconds, otherwise within 10 seconds.

Solution

Define the following predicates, each of them has the variable *t* as argument, P(t) meaning that P is true at the instant t:

MO:	the system is off
OpOn:	the system is on and the OS has been loaded
ApOn:	the system is on and the application A has been loaded
Pow:	the power switch is pressed
SH:	the SHIFT key is pressed

The following formula specifies what the exercise asked (not everything that should be specified in the real case):

$\forall t\ ((MO(t) \wedge Pow(t) \wedge \forall t'\ (t \leq t' \leq t+3 \rightarrow \neg SH(t')) \rightarrow \quad \exists\, t''\ (t+3 < t'' \leq t+15 \wedge OpOn(t'') \wedge ApOn\ (t''))\ )\ ) \wedge$

$MO(t) \wedge Pow(t) \wedge \exists\, t'\ (t < t' \leq t+3 \wedge SH(t')) \rightarrow \exists\, t''\ (t+3 < t'' \leq t+10 \wedge OpOn(t''))\ )\ )$

Si noti che, anche se non indicato esplicitamente nella definizione informale, l'istante t'' in cui la macchina deve essere disponibile, è seguente a t+3, poiché entro quel tempo l'utente ha margine per decidere se vuole premere SHIFT.

Notice that, even if it isn't in the informal definition, the instant t'' in which the system has to be available, is after t+3, because before that the user can decide if press or not SHIFT.

The definition above (both in the original formulation and in the formalization) leaves some open question. For example:

- What happens if the power switch is clicked when the system is already on?
- It is a correct behavior to load the application A anyway, and then in case unload it, provided it happens within 10 seconds? For what indicated in the requisites yes, but in practice this behavior will not be accepted by the users.
- It is correct to boot the system, load the OS and the application within 12 seconds and then halt the system? In this case too, this behavior is formally correct, but it will not be accepted by the users.

Exercise 1.6.11

Specify using an appropriate pair of <pre-, post-condition> a program or a program fragment that, given two positive integers and put z=1 if they are relatively prime, z=0 otherwise.

Notice that, for simplicity we assume that the data x and y and the result z are stored in memory cells: there is no interest in the operations of input / output.

Solution

Pre: $x \in N \wedge y \in N \wedge x>0 \wedge y>0$

$\mathrm{Pri}(x,y) \leftrightarrow (\neg \exists\, k,h_1,h_2\ (k,h_1,h_2 \in N \wedge k>1 \wedge h_1*k = x \wedge h_2*k = y))$

Post: $((z=1 \wedge \mathrm{Pri}(x,y)) \vee (z=0 \wedge \neg\, \mathrm{Pri}(x,y))$

Exercise 1.6.12

Give a formal description of the following rules about opening a safe, using any appropriate formalism:

There are two different keys belonging to two different employees. The safe has to open automatically only if the two keys are inserted in their keyholes before a given amount of time Δ, during which the safe must be closed. Once opened, the safe automatically closes itself after exactly K time units.

For the sake of simplicity, you can assume that opening and closing are two instant events.

Solution

Formalize the problem using first order logic formulas, assuming a continuous temporal domain. Other formalizations based on operational formalisms could be correct as well.

Use the following predicates:

- InsertK1(t): key 1 is inserted at time t;
- InsertK2(t): key 2 is inserted at time t;
- Open(t): safe opens at time t;
- Close(t); safe closes at time t;
- Opened(t): safe is opened at time t;
- Closed(t): safe is closed at time t.

The following formulas describe the correct behavior of the safe.

The safe cannot open and close at the same time.

$\mathrm{Open}(t) \rightarrow \neg\, \mathrm{Close}(t)$

$\mathrm{Close}(t) \rightarrow \neg\, \mathrm{Open}(t)$

The safe opens at time t if and only if a key is inserted at the same time t and the other key has already been inserted within Δ time instants during that the safe has always been closed.

$\mathrm{Open}(t) \leftrightarrow$
$(\mathrm{InsertK1}(t) \wedge \exists\, t_1(\mathrm{InsertK2}(t_1) \wedge t\text{-}\Delta \le t_1 \le t \wedge (\forall t_2\ (t_1 \le t_2 \le t \rightarrow \mathrm{Closed}(t_2))))) \vee$
$(\mathrm{InsertK2}(t) \wedge \exists\, t_1(\mathrm{InsertK1}(t_1) \wedge t\text{-}\Delta \le t_1 \le t \wedge (\forall t_2\ (t_1 \le t_2 \le t \rightarrow \mathrm{Closed}(t_2)))))$

The safe closes after it has been opened exactly K time units.

$\mathrm{Close}(t) \leftrightarrow (\forall\, t_2\ (t - K < t_2 < t \rightarrow \mathrm{Opened}(t_2))$

The safe is opened at time t if and only if there is a past instant in that it has been opened and after that it has never been closed.

$\mathrm{Opened}(t) \leftrightarrow \exists\, t_1\ (t_1 < t \wedge \mathrm{Open}(t_1) \wedge (\forall\, t_2\ (t_1 < t_2 \le t \rightarrow \neg\, \mathrm{Close}(t_2))))$

$\mathrm{Closed}(t) \leftrightarrow \exists\, t_1\ (t_1 < t \wedge \mathrm{Close}(t_1) \wedge (\forall\, t_2\ (t_1 < t_2 \le t \rightarrow \neg\, \mathrm{Open}(t_2))))$

This formal description is correct for any state of the safe but the initial one. In fact, if the safe is now closed, it has to exists an instant in the past in which it has been opened (because the open state is caused by the event Close, which can occur only if the safe has been opened for a time interval K). Besides if the safe is now opened it has been necessarily closed in the past, because the Opened state is caused by the Open event, that requires that the safe was previously closed.

To solve this problem it is possible to add an initialization, for example assuming that the automatic closing behavior has been activated in a specific time t_0:

$\mathrm{Close}(t) \leftrightarrow (t \geq t_0+k \wedge \forall\, t_2\ (t - K < t_2 < t \rightarrow \mathrm{Opened}(t_2))$

$\mathrm{Opened}(t) \leftrightarrow (\exists\, t_1\ (t_1 < t \wedge \mathrm{Open}(t_1) \wedge (\forall\, t_2\ (t_1 < t_2 \leq t \rightarrow \neg\, \mathrm{Close}(t_2))))) \vee t < t_0+k)$

Exercise 1.6.13

A transmission channel reveives as input a signal *i* at in some time instants and has to produce an output signal *o* accordingly with the following rule: each time it receives *i* within k time intervals from the latest signal received, it has to produce the output signal *o* in h time intervals from the latest *i* signal received.

1. Formalize that rule choosing the most suitable formalism. It is better to assume a dense time domain (Rational or real numbers), but the choice od a discrete time domain (Integers) is also admitted.
2. Enrich the rule by adding the following clause: "unless *in the meanwhile* it is not received as input a *i*^ cancel signal."

Solution to point 1.

Being T the time axis (That could be the set of the integer, of the rational or of the real numbers), the *i(t)* predicate states the fact that at the time instant *t* the input signal *i* is received. The *o(t)* predicate has a similar meaning referring to the *o* signal.

Therefore the requirement on the channel is formalized as follows:

$\forall\, t_1,t_2 \in T\ (((i(t_1) \wedge i(t_2) \wedge t_1<t_2\leq t_1+k) \rightarrow \exists\, t_3 \in T\ (o(t_3) \wedge (t_3\leq t_2+h)))$

Solution to point 2.

The *in the meanwhile* clause is a bit ambiguous: it could refer both to the interval between the two *i* signals or to the interval between the reception of *i* and the emission of *o*. The latter interpretation seems to better reflect the actual requirement and it is less trivial to formalize. A formalization of the latter interpretation is the following:

$\forall\, t_1, t_2 \in T\ ((i(t_1) \wedge i(t_2) \wedge t_1<t_2\leq t_1+k) \rightarrow$

$(\forall\, t_3\ ((t_1 \leq t_3 \leq t_2\ ) \rightarrow \neg\, i\text{^}(t_3)) \rightarrow \forall\, t_4\ ((t_2 \leq t_4 < t_2+h) \rightarrow (\neg\, i\text{^}(t_4) \wedge \neg\, o(t_4))) \rightarrow o(t_2+h)$

The formula states that the output *o* has to be emitted when two *i* signals are received in a row, it has not been emitted in *h* time instants from the second occurrence of *i* and in the meanwhile the cancel signal *i*^ has not been received in *h* time instants from the first occurrence of *i*. The *o* signal has to be emitted at the time instant $t_2 + h$.

Notice that this does not interfere with the emission of the *o* signal before that time and even if such conditions are missing. For instance it could be emitted without *i* or even if *i*^ is received. The formula only specify the condition that turns into the emission of the output signal without excluding the other possibilities.

Exercise 1.6.14

Using a first order formula, give a formal description of the following sentence:

"If a problem P is decidable then it is also semi-decidable, but if it is semi-decidable and its complement is not decidable, then P is not decidable. On the contrary, if P is not semi-decidable its complement is not decidable"

Solution:

Used symbols:

P: variable symbol, referring to a generic problem

Comp: function symbol, referring to the complement of a problem

Dec: predicate symbol, indicating whether the problem is decidable

SemDec: predicate symbol, indicating whether the problem is semi-decidable

$$\begin{aligned}\forall P\quad & (Dec(P) \rightarrow SemDec(P)) \wedge \\ & (SemDec(P) \wedge (\neg Dec(Comp(P)) \rightarrow \neg Dec(P)) \wedge \\ & (\neg SemDec(P) \rightarrow \neg Dec(Comp(P)))\end{aligned}$$

Exercise 1.6.15

Consider an electric device running on battery, initially charged and not rechargeable.

The device is initially off. It can work in 2 modes (1 and 2) that consume different amount of power: in mode 1 the power consumption allows work for a period T, while in mode 2 the power consumption is doubled. The event CM causes switching from a mode to the other (if the device was working in mode 1, it will work in mode 2, and vice-versa).

At time t_0 the device is turned on, and it starts working in mode 1. the device turns off only when the battery runs out. Specify the device behavior.

Solution

States: OFF(t), ON(t,K,t_c). In the ON state K indicates the mode and can be 1 or 2. t_c indicates the charge state in terms of residual working time in mode 1.

Event: CM(t) indicates the mode switch.

$\exists\, t_c \wedge t_c>0 \wedge (ON(t,1,t_c) \vee ON(t,2,t_c)) \rightarrow \neg OFF(t)$

$OFF(t) \rightarrow \neg\exists\, t_c \wedge t_c>0 \wedge (ON(t,1,t_c) \vee ON(t,2,t_c))$

$OFF(t) \vee (\exists\, t_c \wedge t_c>0 \wedge (ON(t,1,t_c) \vee ON(t,2,t_c)))$

$\forall t_c\ (\neg\exists\, K\ (ON(t,K,t_c) \wedge K\neq 1 \wedge K\neq 2)$

$ON(t_0,1,T) \wedge \forall t_1\ (t_1<t \rightarrow OFF(t))$

$ON(t,1,t_c) \wedge t_c>0 \wedge \neg CM(t) \rightarrow \exists\, t_2\ (t<t_2\leq t+t_c \wedge (\forall t_3\ (t\leq t_3\leq t_2 \rightarrow \neg CM(t_2) \wedge ON(t_3,1,t_c-(t_3-t))))$

$ON(t,2,t_c) \wedge t_c>0 \wedge \neg CM(t) \rightarrow \exists\, t_2\ (t<t_2\leq t+t_c/2 \wedge (\forall t_3\ (t\leq t_3\leq t_2 \rightarrow \neg CM(t_2) \wedge ON(t_3,2,t_c-2(t_3-t))))$

$ON(t,1,t_c) \wedge t_c>0 \wedge CM(t) \rightarrow \exists\, t_1\ (t<t_1\leq t+t_c/2 \wedge \forall t_2\ (t<t_2\leq t_1 \rightarrow \neg CM(t_2) \wedge ON(t_2,2,t_c-2(t_2-t))))$

$ON(t,2,t_c) \wedge t_c>0 \wedge CM(t) \rightarrow \exists\, t_1\ (t<t_1\leq t+t_c \wedge \forall t_2\ (t<t_2\leq t_1 \rightarrow \neg CM(t_2) \wedge ON(t_2,1,t_c-(t_2-t))))$

$(ON(t,1,0) \vee ON(t,2,0)) \rightarrow \forall t_1\ (t_1 \geq t \rightarrow OFF(t))$

Exercise 1.6.16

Using a first order formula, give a formal description of the following rules:

Rule 1: If a driver commits an offence, a fine will be notify within 150 days from the date of offence, and the driver will pay it within 30 days since the notification. It is suggested to use predicates of the type Offence(A,t) indicating the fact that the driver A has committed an offence in the day t. For the sake of simplicity you can assume that the driver will not commit more than one offence.

Rule 2: If a driver commits an offence and the fine is notified to him/her within 150 days from the date of the offense, he/she must pay the fine within 30 days from the date of the notification. In case that the driver will not pay within 30 days, he/she will be tried. Use the same hint given for the rule 1.

Solution

con l'ovvio significato del nuovo predicato Process.

Rule 1:

The predicate Offence is defined as suggested in the exercise. The predicates Notify(A,t) and Pay(A,t) are defined in a similar way. Using these predicates the rule is formalized by the following formula:

$\forall A, t((Offence(A,t) \rightarrow \exists t_1, t_2((Notify(A,t_1) \wedge t \leq t_1 \leq t+150) \wedge (Pay(A, t_2) \wedge t_1 \leq t_2 \leq t1+30))$

Rule 2:

$\forall A,t,t_1,t_2((Offence(A,t) \wedge Notify(A,t_1) \wedge t \leq t_1 \leq t+150 \wedge \forall t_2\ (t_1 \leq t_2 \leq t1+30 \rightarrow \neg Pay(A,t_2)))\rightarrow$

$\exists t_3(Tried(A, t_3) \wedge t_1+30<t_3))$

with the predicate Tried defined trivially as the others

Exercise 1.6.17

Formalize with a first order formula the following rules:

Rule 1: a student cannot attend a course taught by a relative.

Rule 2: the rule above is modified and detailed in this way: a student cannot attend a course taught by a relative with a relationship degree < 7. The relationship degree is defined as follows: if X is parent of Y, then X and Y are relatives of degree 1. if X is relative of degree *g* with Y and X is parent of Z, then Y is relative of degree *g+1* with Z. For each degree the relation is symmetric.

Solution

Rule 1

Use the following predicates:
Attend (stud, c) (The student stud attends the course c)

Relative (x, y) (The person x is relative of the person y; the relation defined by the predicate is symmetric)
Teaches(prof, c) (Professor prof teaches the course c)
Obviously students and teacher are persons.
The following formula then formalizes the rule 1:

$\forall$stud, prof, c ((Teaches(prof, c) $\wedge$ Relative(prof, stud) $\rightarrow \neg$ Attend (Stud, c))

Regola 2

With respect to *Rule 1*, we have to modify the predicate Relative: Relative(x, y, g) indicates that x and y are relatives of degree g.
The semantics of Relative(x, y, g) is defined bu the following additional rules, after we introduce the new elemental predicate Parent(x, y) (x is parent of y):

$\forall$x, y, z, g (Parent(x,y) $\rightarrow$ Relative(x, y, 1) $\wedge$Relative(x,y,g) $\rightarrow$ Relative(y, x, g) $\wedge$Relative(x,y,g) $\wedge$ Parent(x,z) $\rightarrow$ Relative(y, z, g+1))

The rule 1 above is then trivially modified.

Exercise 1.6.18

Formalize with first order formulas the following requirements on an airport traffic:

1. Any pair of airplanes does not have to be a distance lower than *k* meters.
2. Un aereo può atterrare o decollare solo dopo almeno z minuti dall'ultimo decollo o atterraggio.
3. An airplane can take-off or land after that at least *z* minutes have passed since the last take-off or landing.

Then modify the first requirement in the following way:

1. Any pair of airplanes cannot be at a distance inferior to *k* meters to each other, unless they are in the parking zone. In that case they must respect a distance of more than *h* meters (h < k).

Solution

Define the following predicates:

Pos (V,P):	the vehicle V is on the point P of the airport
Land(V,t):	the vehicle V lands at time t
TakeOff(V,t):	the vehicle V take-off at time t

Dist (P1, P2): function that indicates the distance between the point P1 and the point P2 of the airport.

The requirements 1 and 2 can then be formalized as follows:

1. $\forall V_1, V_2, P_1, P_2\ ((Pos\ (V_1,P_1) \wedge Pos\ (V_2, P_2) \wedge (V_1 \neq V_2)) \rightarrow Dist\ (P_1, P_2) \geq k)$
2. $\forall V_1, V_2, t_1, t_2\ ((TakeOff(V_1,t_1) \vee Land(V_1,t_1)) \wedge (TakeOff(V_2,t_2) \vee Land(V_2,t_2)) \wedge (V_1 \neq V_2)) \rightarrow |t_1 - t_2| \geq z)$

To formalize the modified first predicate, there is the need to introduce another predicate:

Park(P): the point P belongs to the parking area

The modifyied requirement could be expressed in the following way:

$\forall V_1, V_2, P_1, P_2$ ((Pos(V_1,P_1) $\wedge$ Pos(V_2, P_2) $\wedge$ ($V_1 \neq V_2$) $\wedge$ ($\neg$ Park(P_1) $\vee$ $\neg$ Park(P_2)) $\rightarrow$ Dist(P_1, P_2) $\geq$ k)

$\wedge$

((Pos(V_1,P_1) $\wedge$ Pos(V_2,P_2) $\wedge$ ($V_1 \neq V_2$) $\wedge$ (Parch(P_1) $\wedge$ Parch(P_2)) $\rightarrow$ Dist(P_1, P_2) $\geq$ h))

or, in an equivalent way:

$\forall V_1, V_2, P_1, P_2$ ((Pos(V_1,P_1) $\wedge$ Pos(V_2, P_2) $\wedge$ ($V_1 \neq V_2$) $\rightarrow$

((($\neg$ Park(P_1) $\vee$ $\neg$ Park(P_2)) $\rightarrow$ Dist(P_1, P_2) $\geq$ k)

$\wedge$

((Park(P_1) $\wedge$ Park(P_2)) $\rightarrow$ Dist(P_1, P_2) $\geq$ h)))

Note:

The introduced predicates are enough descriptive to formalize the requirements requested in the exercise. However for the formalization of a more complete set of requirements on the behavior of the traffic on an airport, it will be probably more appropriate to use predicates containing more information, such as Pos(V,P,t) to assert that the vehicle V is on the point P at time t.

Exercise 1.6.19

Write a first order formula defining the fact that two numbers are relatively prime.

Solution

Relatively_prime(n,m) $\leftrightarrow$ ($\neg$($\exists$ x, y , z(x > 1 $\wedge$ n = x*y $\wedge$ m = x*z)))

Exercise 1.6.20

Write a first order formula, defining the language of the strings that:

- start with the letter *'a'* and end with the letter *'b'*, or
- contain at least three *'c'* in a row.

Consider the strings defined on the {a,b,c} alphabet.

Solution

$\forall$ x(x $\in$ L $\leftrightarrow$ (($\exists$y (y$\in V_T^*$) $\wedge$ (x = ayb)) $\vee$ ($\exists$ z, y (z$\in V_T^*$) $\wedge$ (y$\in V_T^*$) $\wedge$ (x = zcccy))))

Exercise 1.6.21

Specify with a first order formula a signal counter. At thebeginning the counter value is 0. Whenever a signal arrives (the arrival of a signal is an isolated event, i.e. it happens in only one instant inside an arbitrary small but not empty time interval) the counter is incremented by 1.

Solution

Supponiamo di considerare il tempo continuo.Si definisca con il simbolo di funzione cont(t) il valore del contatore all'istante t; con S(t), l'emissione del segnale all'istante t. La specifica è allora formalizzata dalla formula seguente:

Assuming that time is continuous, the function *cont(t)* returns the counter value at time *t* and *S(t)* represents the emission of the signal at time *t*. The specification is then formalized as follows:

$$\text{Event\_S} \wedge$$
$$\exists t_0 (\forall t_1 (t_1 \le t_0 \rightarrow cont(t_1 = 0) \wedge \neg S(t_1))$$
$$\wedge$$
$$\forall t, k((t \ge t_0 \wedge k > 0) \rightarrow$$
$$(cont(t){=}k \leftrightarrow (\exists t_1 (\text{Up\_to\_now\_cont}(t_1) = \text{k - 1}) \wedge S(t_1) \wedge \forall t_2 (t_1 < t_2 \le t \rightarrow \neg S(t_2))))))$$

where, as usual, Event_S is a short hand for :

$$\forall t (S(t) \rightarrow \exists \delta (\forall t_1 (t - \delta < t_1 < t \vee t < t_1 < t_1 + \delta) \rightarrow \neg S(t_1)))$$

and Up_to_now_cont(t) is a short hand for :

$$\exists \delta (\forall t_1 (t - \delta < t_1 < t) \rightarrow S(t_1)) \qquad (\delta > 0)$$

Exercise 1.6.22

Formalize, using any appropriate formalism, the rules of the game "TRIS": the two players at turn put a symbol in a cell of a conceptually infinite Cartesian plane. Each cell is identified by his integer coordinates. Player 1 uses the symbol '*' while player 2 fills the cells with the symbol 'o'. The first player who manages to form a line in the plane (horizontal, vertical or diagonal) of at least 3 (or, more in general, *k*) consecutive symbols of his type, wins the game.

Note that the request is the formalization of the game rules, not the development of a winning strategy.

Solution

A possible formalization of the TRIS game could be defined with a state machine, whose configuration consists of:

1) a finite state space Q = {q*, qo, q*w, qow}, where q* indicates that the next move belongs to player 1; qo indicates that the next move belongs to player 2; q*w indicates that player 1 won; qo w indicates that player 2 won.
2) A function s: Z X Z ---> {*, o, b} (Where b indicates a blank space)

The transition relationship between 2 configurations c_1 and c_2 is defined as follows:

c_1 |--- c_2 if and only if

$c_1 = <q_o, s_1>$, $c_2 = <q^*, s_2>$ and

$\exists i, j((s_1(i, j) = b \wedge s_2(i, j) = o) \wedge$
$\forall h,k(< h,k > \neq < i, j > \rightarrow s_1(h,k) = s_2(i, j))) \wedge$
$\neg\exists m,n(inARow\ (m,n,3,o))$

Where the *inARow(m,n,k,o) predicate is a short hand for:*

$\forall p((m \leq p < m+k) \rightarrow s_2(p,n) = o) \vee$
$\forall p((n \leq p < n+k) \rightarrow s_2(m, p) = o) \vee$
$\forall p, r(((m \leq p < m+k) \wedge (n \leq r < r+k) \rightarrow s_2(p,r) = o)$

or

c_1 = <q*, s_1>, c_2 = <qo, s_2> and
$\exists i, j((s_1(i, j) = b \wedge s_2(i, j) = *) \wedge$
$\forall h,k(< h,k > \neq < i, j > \rightarrow s_1(h,k) = s_2(i, j))) \wedge$
$\neg\exists m,n(inARow\ (m,n,3,*))$

or

c_1 = <q*, s_1>, c_2 = <q*w, s_2> and
$\exists i, j((s_1(i, j) = b \wedge s_2(i, j) = *) \wedge$
$\forall h,k(< h,k > \neq < i, j > \rightarrow s_1(h,k) = s_2(i, j))) \wedge$
$\exists m,n(inARow\ (m,n,3,*))$

or

c_1 = <qo, s_1>, c_2 = <qow, s_2> and
$\exists i, j((s_1(i, j) = b \wedge s_2(i, j) = o) \wedge$
$\forall h,k(< h,k > \neq < i, j > \rightarrow s_1(h,k) = s_2(i, j))) \wedge$
$\exists m,n(inARow\ (m,n,3,o))$

The initial configuration of the machine is the one in which the state is q* and s_0(i,j) = b for all i,j.

Final states are q*w and qow.

Exercise 1.6.23

Give a formal description of the following rules about the access to a telephone box:

- When a person arrives, if the box is occupied he/she cannot enter;
- when a person arrives and the box is free, he/she enters and must exit within time k.

It is preferred, but not strictly necessary, that the rules are formalized referring to a *continuous* time model, instead of using a *discrete* time model.

Solution

Defining a generic predicate P as an *event*, the following property is true:

$P(t) \dashrightarrow \exists\, \delta |\, (\forall\, t_1,\, t - \delta \leq t_1 < t \wedge t < t_1 \leq t + \delta \Rightarrow \neg\, P(t_1))$.

In the same way, the definition of P as a *state* and assuming a duration conventionally closed on the left and opened on the right, the following property is true:

$P(t) \Rightarrow \exists\delta |\, ((\forall\, t \leq t_1 < t + \delta \Rightarrow P(t_1)$.

After that premises, it is possible to define the following predicates:

Arrives(t):	a person arrives at time t	(event)
Occupied(t):	the box is occupied at time t	(state) (Free(t) is trivially its negation)
Enter(t):	a person enters at time t	(event)
Exit(t):	a person exits at time t	(event)

For every state S, define "UptonowS(t)" as an short hand for: $\exists\delta |\, (\forall\, t_1,\, t - \delta \leq t_1 < t \Rightarrow S(t_1))$.

and "BecomesS(t)" as the short hand for: $\exists d |\, (\forall\, t_1,\, t \leq t_1 < t + \delta \Rightarrow S(t_1))$.

Now the rules of the service can be formalized in the following way:

$\text{Arrives}(t) \wedge \text{UptonowFree}(t) \Rightarrow \text{Enter}(t)$

$\text{Enter}(t) \Rightarrow \text{BecomesOccupied}(t)$

$\text{Enter}(t) \Rightarrow \exists t_1 |\, \text{Exit}(t_1) \wedge t < t_1 \leq t + k$

$\text{UptonowOccupied}(t) \wedge \neg\, \text{Exit}(t) \Rightarrow \text{Occupied}(t)$

$\text{UptonowOccupied}\ (t) \wedge \text{Exit}(t) \Rightarrow \text{BecomesFree}(t)$

$\neg\, \text{Arrives}(t) \wedge\ \text{UptonowFree}(t) \Rightarrow \text{Free}(t)$

The given formalization can be generalized in several ways: for instance, assuming that H telephone boxes are exclusively dedicated to interurban calls and H telephone boxes are exclusively dedicated to urban calls, it would be possible to write the following formulas:

$\text{ArrivesUrban}(t) \wedge \text{UptonowOccupiedUrban}(h, t) \wedge h < H \Rightarrow \text{EnterUrban}(t)$

$\text{EnterUrban}\ (t) \wedge \text{UptonowOccupiedUrban}\ (h, t) \wedge h < H \Rightarrow \text{BecomesOccupied}(h + 1, t)$

giving to the new symbols their obvious meaning.

Exercise 1.6.24

An *event* is something that happens in a certain amount of time (the time is considered continuos) and does not happen for a period of time subsequent to its occurrence. As *state* we mean a situation that has a beginning and an end, and went on for a nonzero interval of time between the beginning and end.

Formalize (preferably by logical formulas) an *event counter*, i.e. a state that at some initial time has the value 0 and its value increases by one each time an event E happens.

It is suggested to formalize the occurrence of the event E by a predicate that is true when it happened. Similarly, another predicate EC could be used to establish that at any given time the value of the event counter is k.

Solution

E(t) states the occurence of E at the time t.

To formalize the fact that E is an event should be established that if E happens at time t, it does not happen at an interval following t:

$\forall t(E(t) \Rightarrow \exists\, \delta\, (\forall\, t_1\, (t \leq t_1 < t + \delta \Rightarrow \neg\, E(t_1)))$

CE(t,k) denotes the fact that at time t the events counter is k (all k values in this context are taken ≥ 0). The behavior of the counter is described by the following formula:

$\exists t_0\, (CE(t_0, 0) \wedge \forall\, t_1\, (t_1 < t_0 \Rightarrow CE(t_1, 0))$

$\forall\, t_1, \forall\, t_2\, \forall\, k \forall\, h\, (CE(t_1, k) \wedge CE(t_2, h)$ and $t_1 < t_2 \Rightarrow k \leq h)$

/* CE is not decreasing */

$\forall\, t\, \forall\, k\, (CE(t, k) \Rightarrow \exists\, \delta\, (\forall\, t1\, (t \leq t1 < t + \delta \Rightarrow CE(t, k)))$

/* This formula requires CE to hold a constant value during left-closed and right-open time intervals */

$\forall\, t\, \forall\, k\, (\, \exists\, \delta\, (\forall\, t_1\, ((t - \delta \leq t_1 < t \Rightarrow CE(t, k) \wedge E(t) \Rightarrow CE(t, k{+}1)))$

/* If the counter value was k for an interval (right-open) until the time t and in t happens E, then the value increases by 1. Instead, the counter value only increases if E happens */

$\forall\, t\, \forall\, k\, (\, \exists\, \delta\, (\forall\, t_1\, ((t - \delta \leq t_1 < t \Rightarrow CE(t, k) \wedge CE(t, k{+}1) \Rightarrow E(t))))$

Exercise 1.6.25

The behavior of a lamp on and off can be described in informal terms as follows:

- when the plug is inserted, the lamp turns on and off alternately with a period Δ.
- When the plug is switched off, the lamp is off.

Formalize the behavior of the lamp with suitable first order logic formulas using the following predicates:

INS(t)	the plug is inserted at time t
DIS(t)	the plug is switched off at time t
SI(t)	at time t the plug is inserted
SD(t)	at time t the plug is not inserted
LA(t)	at time t the lamp is on
LS(t)	at time t the lamp is off.

The formulas must specify the predicates SD, LA, LS starting from the INS and DIS predicates, which have to be considered as events, i.e. predicates true in isolated points of the time axis.

An example of the lamp behavior, in which holds H, K < Δ, is shown in Figura 1-96.

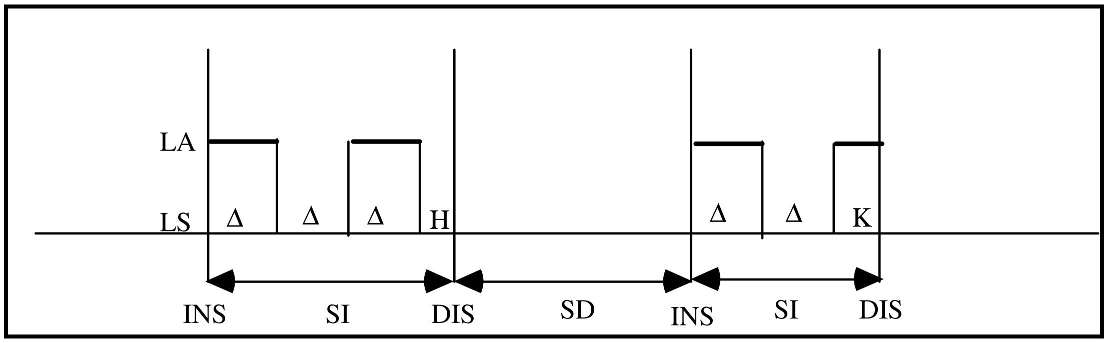


Figura 1-96 An example of the lamp behavior.

Solution

The behavior of the lamp is specified by the following formulas (implicitly universally quantified over time t and with *k* as an integer variable):

$SI(t) \Leftrightarrow \exists t_o\, (t_o \leq t \wedge INS(t_o) \wedge \forall t_1\, (t_o \leq t_1 \leq t \Rightarrow \neg\, DIS(t_1)))$

$SD(t) \Leftrightarrow \neg\, SI(t)$

$LA(t) \Leftrightarrow \exists t_o\, (\; t_o \leq t \wedge INS(t_o)$

$\wedge\, \forall t_1\, (\; t_o \leq t_1 \leq t \Rightarrow SI(t)\;)$

$\wedge\, \exists k\, (k \geq 0 \wedge (k\ \mathbf{mod}\ 2 = 0) \wedge (k \cdot \Delta \leq t - t_o < (k+1) \cdot \Delta)))$

$LS(t) \Leftrightarrow \neg\, LA(t)$

Exercise 1.6.26

Formalize through the following formula of first-order, the following condition relating to the behavior of a driver's car:

"As soon as the vehicle you are driving reaches a distance of K meters from the vehicle in front, the driver must stop within 1/2 second"

It is suggested to use the following predicates:

D(m,t) which states that the distance between the vehicle and the one preceding at time t it is m;

F(t) which states that the brake of the vehicle is pressed at time t;

Solution

$\forall\, t$

$(D\, (m, t) \wedge m \leq K \wedge$

$(\exists\, \Delta\, ((\Delta > 0) \wedge$

$(\forall\, t_1\, (t - \Delta \leq t_1 < t) \Rightarrow \exists\, m_1\, (D\, (m_1, t_1) \wedge m_1 > K)))$

$\Rightarrow$

$\exists\, t_2\, (F(t_2) \wedge t \leq t_2 \leq t + 1/2)$

Exercise 1.6.27

Formalize using a first-order formula that defines the following property of opening condition of a safe:

"The safe opens if and only if the two different keys are both inserted in an interval of at most k seconds. The keys cannot be insterted at the same time"

Solutions

Using the following predicates,

O(t): the safe opens at time t

I1(t): key 1 is inserted at time t

I2(t): key 1 is inserted at time t

the property is described as follows:

$\forall t\ (O(t) \leftrightarrow \exists\ t_1, t_2\ (t_1 \leq t \wedge t_2 \leq t \wedge |t_1 - t_2| \leq k \wedge t_2 \neq t_1 \wedge I1(t_1) \wedge I2(t_2)))$

Exercise 1.6.28

Describe with a suitable formal model the following requirement on a canne for signal transmission:

The channel receives as input a finite sequence of alphabetic characters terminated by the special character *'$'*. The channel has to produce as output another *'$'* terminated sequence in which all the elements of the input string are present only once. In practice the channel removes the repeated elements.

For instance, the sequence "ababbhjjeuuabf$" could be transformed "abhjeuf$" or in "jeuabhf$".

Notice that the exercise is about the specification of the requirement, not about the implementation of the channel described. A machine that removes the repeated characters would be a suitable implementation but not a satisfying formal formulation of the requirement.

Solution

It is possible to define, in a logic formulation, the character sequence as a function N $\rightarrow$ CHAR (Composed by the set of alphabetic characters plus the terminator *'$'*) that is defined by the following formula:

(•) $\exists\ n\ (\ in\ (n) = \text{'\$'} \wedge \forall\ j\ (j < n \Rightarrow in(j) \neq \text{'\$'}))$

(The constraint on the value of *in* for j > n is deliberatively not specified)

The output sequence is then formalized with a formula which satisfies a requirement similar to the one already specified with the addition of the constraint on the repeated values. In the formula *m* represents the length of the output, that is the index at which the string contains the *'$'* symbol.

(••) $\forall\ j\ (j < n \Rightarrow \exists\ i\ in(j) = out(i)) \wedge \forall\ j\ (j < m \Rightarrow \exists\ i\ in(i) = out(j))$

$\forall\ j,i\ ((j < m \wedge i < m \wedge i \neq j) \Rightarrow out(j) \neq out(i))$

Notice that, the existential quantifier ∃ n reported in formula (•) has to contain in its *scope* the whole formula (••) in order to give meaning to the *m* symbol it contains.

An alternate, more operational, specitication could be describedby an abstract machine associating to the input sequence the set of all the possible output sequences that satisfies the given requirement.

1.7 Miscellanei

Exercise 1.7.1

Write a context-free grammar for the language L of the polynomials without the constant term, with brackets, and in the variable x. The phrases of L are, for example: (1 x ^ 2) + 20x ^ 8 - (10x ^ 2) (the degrees of monomials are not necessarily all distinct). Define, from the grammar, a NPDA that recognizes the language.

Solution

A grammar for L is for instance G = ({0,1,2,3,4,5,6,7,8,9,x,+,^,(,)}, {C, D, N, M, S}, P, S), with the set P of the productions as follows:

S → (S) | M + S | M - S| M | -M

M → DN x^ DN | (M)

N → CN | ε

C → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

D → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A derivation of 20x^3 + 9x^2 is:

S ⇒ M + S ⇒ M + M ⇒ DNx^DN + M ⇒ DCNx^DN + M ⇒ DCx^DN + M ⇒ DCx^D + M ⇒ 2Cx^D + M ⇒ 20x^D + M ⇒ 20x^3 + M ⇒ ... ⇒ 20x^3 + 9x^2.

To build a NPDA from the grammar, you can proceed in a systematic way by defining an automaton A = ({q0, q1, qF}, {Z0} VN VT, VT, δ, q0, Z0, {qF}) , whose transitions are constructed according to the following rules[1]:

1. an ε-move from q0 to q1 that writes the symbol *S* at the top of the stack: δ (q0, ε, Z0) = {<q1, SZ0>}.
2. a move for each production of type A → α: <q1, α> δ (q1, ε, A).
3. a move that consumes every input terminal *a* deleting the same terminal from the top of the stack: δ (q1, a, a) = {<q1, ε>}.
4. a move that brings from q1 to qF when on the stack there is only Z0: δ (q1, ε, Z0) = {<qF, ε>}.

The NPDA corresponding to the grammar G is shown in Figure 1-97.

[1] In such way, the alphabets of the stack symbols and the input symbols are not disjoint. However, it is easy to make them disjoint by marking the elements of V_T

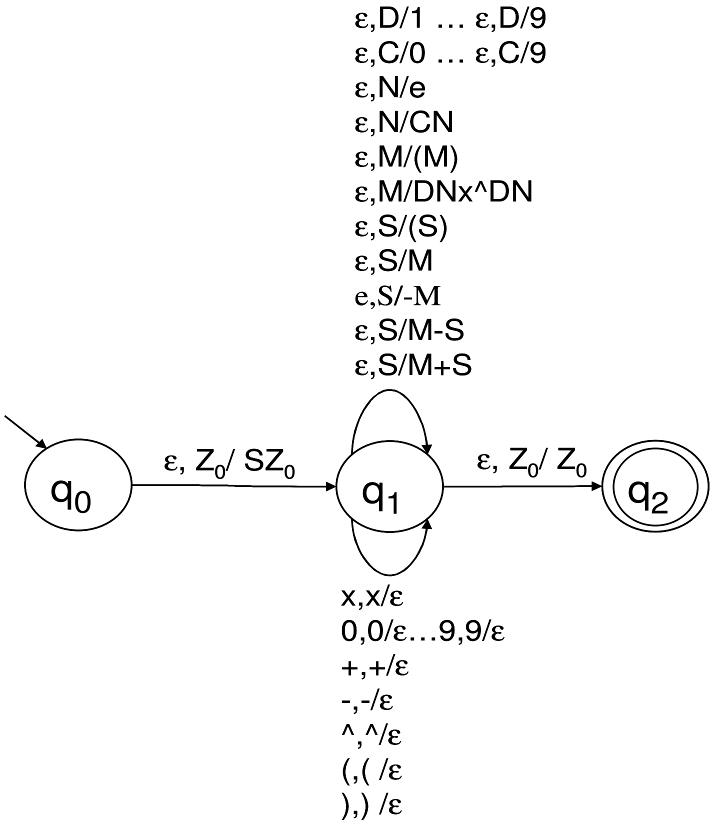


Figure 1-97 A NPDA corresponding the grammar G.

Exercise 1.7.2

In some cases it can be useful to use pushdown automatons that accept a language not through using final (acceptation) states, but through being in a condition in which the stack is empty at the end of the scan of the input string. For instance, such an automaton would be able to recognize the language $\{a^n b^n | n \geq 1\}$ by deleting the initial symbol Z_0 when reading the first *a*, putting in the stack as many *A* as the read *a*, and popping from the stack an *A* for each *b* read and being with an empty stack exactly after the reading of the last *b*.

1. Provide a formal definition for a stack automaton that recognizes on an empty stack without using final states. Formalize as well the language recognition performed by such automaton.
2. Build a pushdown automaton, possibly deterministic, accepting when the stack is empty, that recognizes the language c= $\{a^n b^m c \mid m \geq n \geq 1\}$
3. Build a pushdown automaton, possibly deterministic, accepting when the stack is empty, that recognizes the language $L_2 = \{a^n b^m \mid n \geq m \geq 1\}$
4. Build two grammars G_1 and G_2, in the least powerful class possible, that generate respectively L_1 and L_2.

Solution to point 1

A pushdown automaton recognizing when the stack is empty is defined like a traditional PDA without the final states set $F \subseteq Q$.

The configuration is defined like for the traditional stack automaton.

A string x is accepted "on empty stack" by a stack automaton if and only if:

$<x, q_0, Z_0> \overset{*}{\vdash} <\varepsilon, q, \varepsilon>$ for any $q \in Q$.

Determinism and nondeterminism are defined as in the traditional pushdown automata.

Solution to point 2

A deterministic stack automaton recognizing L_1 on empty stack could operate in the following way:

1. it puts on the stack as many *A* as the *a* read (keeping the initial Z_0 on the bottom of the stack);
2. when it reads *b*, it removes from the stack an *A* for each *b* read, until it reaches Z_0;
3. when it reads additional *b*, if there are any, it keeps the stack as is (i.e. only Z_0);
4. when it reads *c* it removes Z_0 from the stack and stops, accepting the string.

Such an automa ton is reported in Figure 1-98. Notice that an equivalent form is obtainable by putting together q_1 and q_2

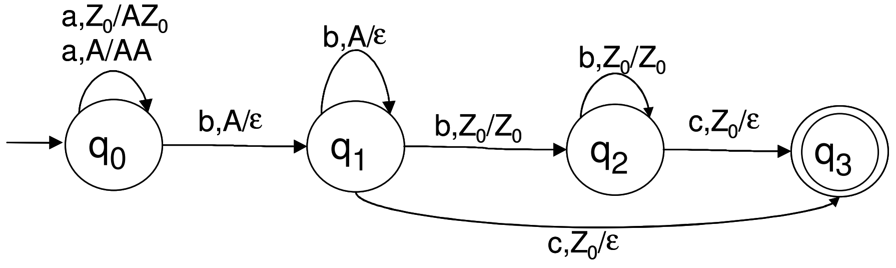


Figure 1-98 A DPDA recognizing L_1 when the stack is empty.

Solution to point 3.

A nondeterministic pushdown automaton recognizing L_2 on empty stack could operate in the following way:

1. it reads an arbitrary number of *a*, keeping the stack as is.
2. nondeterministically, after having read some *a*, it goes on a state where it starts recognizing the language $\{a^nb^n | n \geq 1\}$.

L2 cannot be recognized deterministically since the automaton must have the stack empty exactly when it terminates scanning the input. Otherwise it would have to perform a certain number of ε-moves to empty the stack; but these ε-moves would introduce a nondeterminism because they could be done before the end of the input scan. In other words, the automaton either starts to put *A* on the stack from the beginning, and in this case it could be with a non-empty stack at the end of the input, or, in the opposite case, it has to choose in a nondeterministic way when to start counting *a*.

Such an automaton is shown in Figure 1-99.

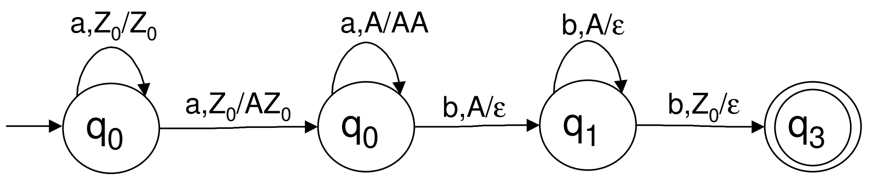


Figure 1-99 A NPDA recognizing L_2 when the stack is empty.

Solution to point 4.

G_1 is defined by the following productions:

$S \to XB$

$X \to aXb \mid ab$

$B \to bB \mid c$

G_2 is defined by the following productions:

$S \to aS \mid A$

$A \to aAb \mid ab$

Exercise 1.7.3

A language can be described using first order formulas defining the strings belonging to it.

For instance $L = a^*b^*$ is defined by the following formula:

$x \in L \Leftrightarrow ((x = \varepsilon) \vee (\exists\, y(x = ay \wedge y \in L) \vee (x = yb \wedge y \in L)))$

Using first order formulas, define the following languages:

$L_1 = a^*b^*c^*$

L_2 = language of palindrome strings in $\{a,b\}^*$. A string is called a palindrome if it can be read the same way from left to right and from right to left. For instance, the string 'aba' is a palindrome.

Solution to point 1.

Let L be the language a^*b^* previously defined. In the same way define $L^\wedge = b^*c^*$. So L_1 is defined by the following formula:

$$x \in L_1 \Leftrightarrow$$
$$((x = \varepsilon) \vee (x \in L^\wedge) \vee (x \in L) \vee$$
$$\exists\, y((x = ay \wedge ((y \in L^\wedge) \vee (y \in L_1))) \vee (x = yc \wedge ((y \in L) \vee (y \in L_1)))))$$

Solution to point 2.

L_2 is defined by the following formula:

$x \in L_2 \Leftrightarrow ((x = \varepsilon) \vee (x = a) \vee (x = b) \vee (\exists\, y(x = aya \wedge y \in L_2) \vee (x = byb \wedge y \in L_2)))$.

Exercise 1.7.4

Build a finite state automaton recognizing the language generated by the R-grammar:

$S \to 1A$ $\quad A \to 0B$ $\quad B \to 0B$ $\quad B \to 0$

Solution

A non-deterministic FSA for a R-grammar can be easily built in the following way:

$Q = V_N \cup \{q_f\}$, $F = \{q_f\}$, $B \in \delta(a,A)$ if and only if $A \rightarrow aB \in P$.

An automaton for the given R-grammar is shown in Figure 1-100. The automaton is not deterministic because it simulates every possible derivation of the grammar.

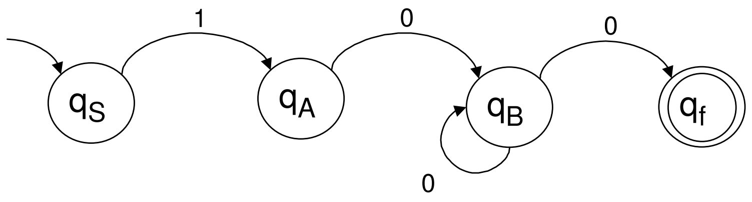


Figure 1-100 A FSA recognizing the language of a R-grammar.

The automaton can be transformed in a deterministic FSA using the mechanical method already used in 0. It is also possible to observ that the given grammar is equivalent to the following:

$S \rightarrow 1A$ $\quad A \rightarrow 0P$ $\quad P \rightarrow 0B$ $\quad B \rightarrow 0B \mid \varepsilon$

From this grammar it is easy to build the automaton shown in Figure 1-101, which is also equivalent to that in Figure 1-100.

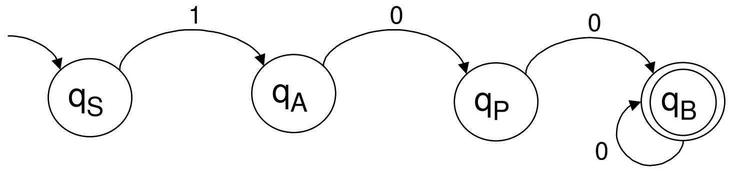


Figure 1-101 A FSA recognizing the language of a R-grammar.

Exercise 1.7.5

Build an abstract machine recognizing the following language:

$L = \{ww^R\}^*$, $w \in \{a, b\}^*$

(remember that w^R indicates the mirror string of w).

Try to use a "minimum power" machine, that is an automaton belonging to a class recognizing the smallest family of languages.

A full specification of the machine is not strictly required: it is sufficient a description precise and clear enough as to leave no doubt about how the automaton is working.

Solution

It is easy to build a non-deterministic pushdown automaton A recognizing L by slightly changing the non-deterministic pushdown automaton A' that recognizes $L' = \{ww^R\}$, being $w \in \{a, b\}^*$.

A' recognizes L' pushing characters of w on the stack and deciding non-deterministically to change state any time in which the symbol on the top of the stack is equal to the symbol read. Afterwards, every symbol read must be equal to the symbol on the top of the stack, which is deleted on every move (if this is not true, A' halts refusing the input string). The string is accepted if a sequence of moves, such that when the reading finishes the stack is empty, exists.

A works like A' with the difference that the special symbol Z_0 (indicating the start of the stack) is kept on the bottom of the stack. When it is reached, A' performs, non-deterministically, one of the following two moves:

a. A' deletes Z_0 from the stack and it goes into a final state. In this way, the input string is accepted if this action is taken exactly at the end of the reading.
b. A' doesn't delete Z_0 from the stack and it goes back to the initial state where the symbols are pushed on the top of the stack.

The same *intuitive* considerations showing that it is not possible to build a pushdown automaton A' recognizing L' deterministically, underlines that it is not possible to build a deterministic pushdown automaton for L. During the reading of the input string x, there is no element, while reading any prefix y, to understand if y is eventually the first half of a string of the type ww^R. So, this choice must be "risked" non-deterministically.

Therefore, a FSA recognizing L cannot exist. So, the class of non-deterministic pushdown automata is the "minimum power" class that is able to recognize L.

Exercise 1.7.6

Formalize the two-stack pushdown automaton, shown in Figura 1-102. It has an input tape, with a reading head, on which is put the input string, two stacks entirely similar to the classic stack automaton stack, and a control unit with a finite number of states. The reading head is initially on the first cell of the input tape, each move depends on the symbol currently under the head, the two symbols at the top of the stack and the current state of the machine, and it consist in doing a movement (to the right or null, in case of ε-move) of the reading head, and substitute the symbols on top of the stacks with strings.

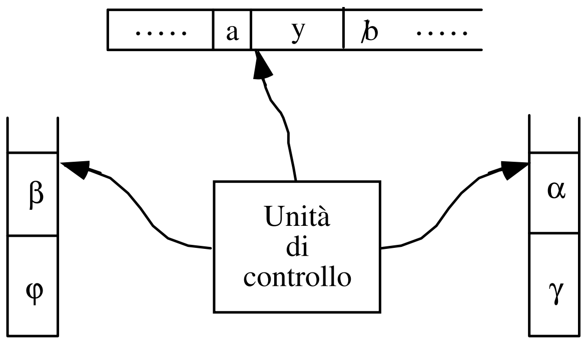


Figura 1-102 A two-stack pushdown automaton.

1. formalize the two-stack pushdown automaton described above, defining in particular the notions of configuration, transitions between configurations, and acceptance of a string by the machine;

2. define the two-stack pushdown automaton recognizing the language $L = \{ww^R \mid w\in\{a, b\}^*\}$;
3. study the device power, in particular with respect to the Turing machine.

Solution to point 1

The 2-Stack Automaton (2SA) is defined by a 6-tuple

$<Q, I, \Gamma, \delta, q_o, Z_o>$

where Q, Γ, q_o, Z_o are defined like the normal stack automaton, I is the input alphabet, including the special character named *blank* and representing a blank space, without any significant symbols, δ is the transition function of type

$$\delta: Q \times (I \cup \varepsilon) \times \Gamma^2 \to Q \times (\Gamma^*)^2$$

and by indicating $\delta(q, i, A, B) = <q', \alpha, \beta>$ we intend that, if we are in the state q, with the symbol i (or ε) under the head and the symbols A and B on the stacks, the machine moves to the state q', moving the head one position to the right (or keep standing if i = ε), and replaces A and B with α and β, respectively. Like we do with the classic stack automaton, to achieve determinism we impose that, for each $q\in Q$, $A,B\in\Gamma$, if $\delta(q, \varepsilon, A, B)$ is defined, then $\delta(q, i, A, B)$ is not defined for each $i\in I$.
A 2SA configuration is represented by the 4-tuple

$$<q, x, \sigma, \rho>$$

in which q is the current state, x is the remaining string (i.e. the fraction of tape that is not blank and not yet passed by the head. The first character of x is under the head), σ and ρ are the strings on the 2 stacks, conventionally called left and right stack.
The transition relation $c \vdash c'$ between 2 configurations is defined this way. $<q, x, \sigma, \rho> \vdash <q', x', \sigma', \rho'>$ if and only if is true, exclusively, one of these 2 conditions:

1. (non-ε-move) We have, as shown in the transition diagram in Figure 1-103, $\delta(q, a, A, B) = <q', \alpha, \beta>$ e

   $x = ay, \sigma = A\gamma, \rho = B\varphi,$

   $\delta(q, a, A, B) = <q', \alpha, \beta>,$

   $x' = y, \sigma' = \alpha\gamma, \rho' = \beta\varphi.$

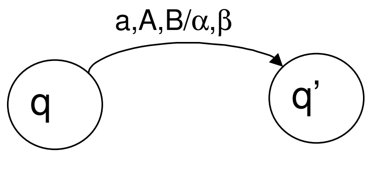


Figure 1-103 Non-ε-move of a 2SA.

2. (ε-move) We have, as shown in the transition diagram in Figure 1-104, $\delta(q, \varepsilon, A, B) = <q', \alpha, \beta>$ e

   $\sigma = A\gamma, \rho = B\varphi,$

   $\delta(q, \varepsilon, A, B) = <q', \alpha, \beta),$

   $x' = x, \sigma' = \alpha\gamma, \rho' = \beta\varphi.$

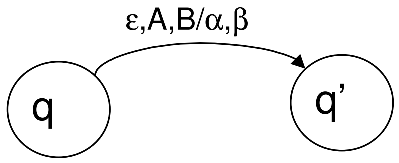


Figure 1-104 ε-move of a 2SA.

Solution to point 2

The language L = $\{ww^R \mid w \in \{a, b\}^*\}$ is accepted by the 2SA shown in Figure 1-105. The automaton copies, in advance, the entire string in the left stack, then it moves repeatedly the string from one stack to the other, deleting the first and last character if they are equal. Adopting the usual convention, we have used the symbol '•' to indicate any element in the stack alphabet. Therefore, in a transition like a,•,Z_0/A•,Z_0, on the left stack will be added the character "A", keeping intact the rest.

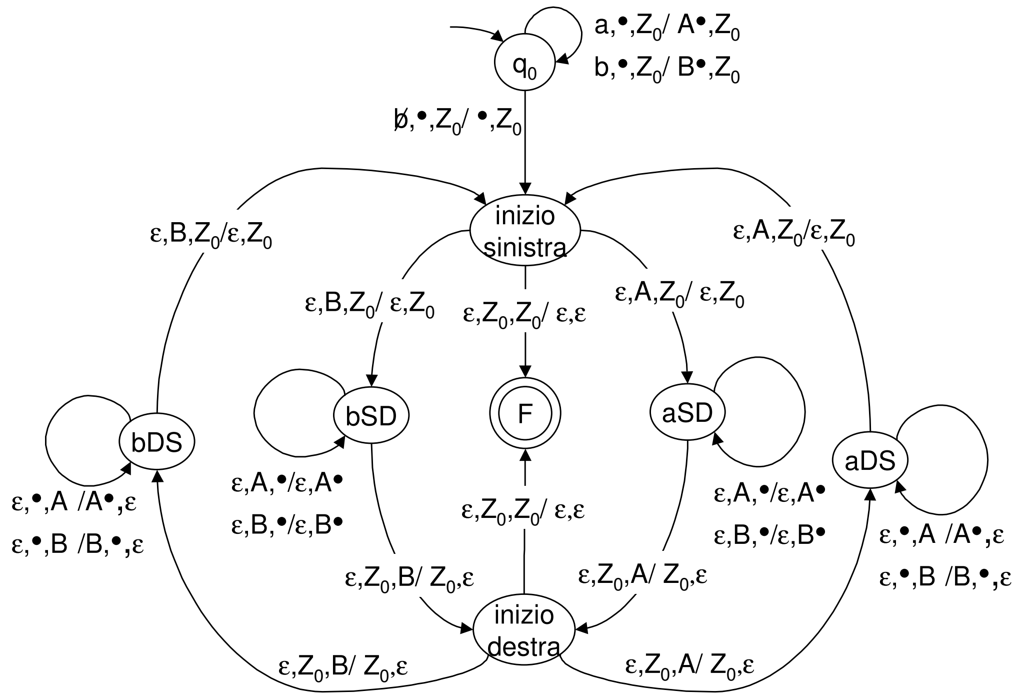


Figure 1-105 2SA L = $\{ww^R \mid w \in \{a, b\}^*\}$.

Solution to point 3

As we can easily understand by confronting the solution to point 2 above with the solution to 1.3.6, the 2SA can easily simulate a single-tape Turing machine, therefore it has the same power as the Turing machine. To prove that, we define a procedure that permits, given any single-tape Turing machine (STM), to build an equivalent 2SA, i.e. that accepts the same language. The 2SA simulates the STM in the following way:

The set of the 2SA states is $Q_{2SA} = \{q_{-2}, q_{-1}\} \cup \{q_p \mid q \in Q_{STM}\}$, where Q_{STM} is the STMset of states. As shown in Figure 1-78, initially the 2SA copies the input string *x* in the left stack (thus going in the configuration <q_{-1}, ε, $x^R Z_o$, Z_o>), then, with a series of ε-moves it moves the content of the left stack in the right stack (going in the configuration $<q_o, \varepsilon, Z_o, xZ_o>$). From here, the 2SA simulates the STM with a sequence of ε-moves, keeping in the left stack the part of the tape to the left of the head, and in the right stack the part of tape to the right of the head plus, on top of it, the symbol under the STM's head. A configuration xqiy of the STM is therefore simulated by the configuration $<q,\varepsilon,x^RZ_o,iyZ_o>$ of the 2SA.

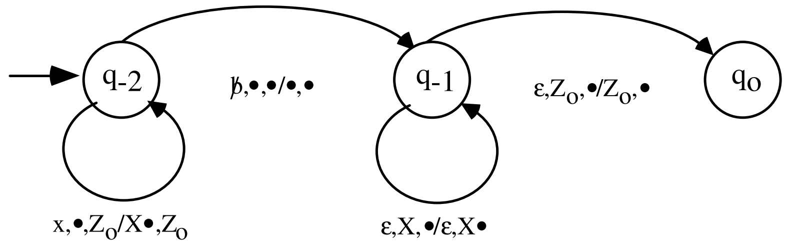


Figure 1-106 Starting moves of a 2SA simulating a STM.

Let's see in details how the STM moves simulation by the 2SA works. Assume that the STM is in the configuration xqiy.

1. If the STM has $\delta(q, i) = <q', c, S>$ and therefore the transition xqiy $\vdash$ xq'cy, the corresponding 2SA has $\delta(q, \varepsilon, A, i) = <q', A, c>$ and therefore the transition $<q, x, \sigma, i\delta> \vdash <q', x, \sigma, c\delta>$. Note: the character A here indicates any character that could be on the stack. Because the STM considers only the character under the head, while the 2SA considers the 2 characters on top of the stacks, we need to indicate that the transition should happen regardless of the content of the left stack (including the case it is empty),Se per la STM si ha $\delta(q, i) = <q', c, L>$ occorre distinguere il caso in cui $x=\varepsilon$ da quello in cui x=sv, con $s\in I^*$ e $v\in I$, per i quali si hanno rispettivamente le transizioni qiy $\vdash$ q'b̸ cy oppure svqiy $\vdash$ sq'vcy. Per la 2SA si hanno rispettivamente le mosse $\delta(q, \varepsilon, Z_o, i) = <q', Z_o, b\!\!/ c>$ e $\delta(q, \varepsilon, A, i) = <q', \varepsilon, Ac>$ per ogni $A\neq Z_o$. Tali mosse danno origine rispettivamente alle transizioni $<q, \varepsilon, Z_o, iyZ_o> \vdash <q', \varepsilon, Z_o, b\!\!/ cyZ_o>$ oppure $<q, \varepsilon, vs^RZ_o, iyZ_o> \vdash <q', \varepsilon, s^RZ_o, vcyZ_o>$.
2. if the STM has $\delta(q, i) = <q', c, L>$, we need to distinguish the case where $x=\varepsilon$ and the case where x=sv, with $s\in I^*$ e $v\in I$, for which we have, respectively, the transitions qiy $\vdash$ q'cy or svqiy $\vdash$ sq'vcy. For the 2SA we have respectively the moves $\delta(q, \varepsilon, Z_o, i) = <q', Z_o, c>$ and $\delta(q, \varepsilon, A, i) = <q', \varepsilon, Ac>$ for each $A\neq Z_o$. These moves originate respectively the transitions $<q, \varepsilon, Z_o, iyZ_o> \vdash <q', \varepsilon, Z_o, cyZ_o>$ or $<q, \varepsilon, vs^RZ_o, iyZ_o> \vdash <q', \varepsilon, s^RZ_o, vcyZ_o>$.
3. if the STM has $\delta(q, i) = <q', c, R>$ we have to distinguish the case where $y=\varepsilon$ or y=ru, with $r\in I$ e $u\in I^*$. for each state $q\in Q_{STM}$, we introduce a prospection state $q_p\in Q_{2SA}$ to see what there is under the top of the right stack, i.e. to verify if $y=\varepsilon$. therefore we have, for the 2SA, $\delta(q, \varepsilon, A, i) = <q_p, cA, \varepsilon>$ and $\delta(q_p, \varepsilon, A, Z_o) = <q', A, Z_o>$ (corresponding to the case $y=\varepsilon$) and $\delta(q_p, \varepsilon, A, B) = <q', A, B>$ for each $A,B\neq Z_o$ (corresponding to the case y=ru). Therefore to the transition xqi $\vdash$ xcq' of the STM corresponds in the 2SA the double transition $<q, \varepsilon, x^RZ_o, iZ_o> \vdash <q_p, \varepsilon, cx^RZ_o, Z_o> \vdash <q', cx^RZ_o, Z_o>$, while to the transition $<q, \varepsilon, x^RZ_o, iruZ_o> \vdash <q_p, \varepsilon, cx^RZ_o, ruZ_o> \vdash <q', cx^RZ_o, ruZ_o>$.

Exercise 1.7.7

Let F be a generic family of automata (for instance: finite automata, pushdown automata, ...). Let F^ be the family of languages not recognized by any automaton in F.
Say, justifying the answer, if F^ is closed under the set operations (union, intersection, complement) in the following cases:

1. F = Finite State Automata
2. F = Pushdown Aautomata

3*. F = Nondeterministic Pushdown Auatomata

4*. F = Turing Machines

Solution

Notice first that the class of languages recognized by machines belonging to F is closed with respect to the complement if and only if the same property is also owned by F^: in fact, if F (better: the class of languages recognized by automata in F) is closed under the complement cannot happen that L belongs to F, and its complement does not belong to it (in this case, the complement would be recognized by an element of F and hence it would be L) and vice versa.

This then responds positively to the question on the complement to the classes of finite automata and pushdown automata deterministic, since the corresponding classes of languages are closed under the complement, and in a negative way for pushdown automata and nondeterministic Turing machines.

None of the considered classes is closed under union and intersection.

For classes closed under the complement F, it is sufficient to identify a language L that is not recognized by an automata in F (certainly existing): then its complement is also in F^, but the intersection and the union of L with its complement are respectively the empty set and the whole universe, both accepted by automata in any of the families considered.

For the NPDA consider the following languages:

$L_1 = \{a^n b^m c^l d^p \mid n,m,l,p \geq 1 \wedge (n=m \vee m=l) \wedge \text{primo}(p)\}$ and

$L_2 = \{a^n b^m c^l d^p \mid n,m,l,p \geq 1 \wedge (n=m \vee m=l) \wedge \neg\text{primo}(p)\}$, both not recognizable by NPDA since to check if a number is prime a TM is needed.

$L_1 \cup L_2 = \{a^n b^m c^l d^+ \mid n,m,l,p \geq 1 \wedge (n=m \vee m=l)\}$ is a non-deterministic context-free language (hence recognizable by NPDA).

$L_1 \cap L_2 = \varepsilon$, recognizable by NPDA (and also by less powerful formalisms).

In conclusion L1 and L2 are both in F^ but their union and their intersection are not.

A similar argument applies to Turing machines taking into account the fact that the existence of languages such that neither the languages nor their complement is recursively enumerable is well known in the literature.

Exercise 1.7.8

Consider the language L defined as follows:

L = { $w_1\$w_2\$\ldots\$w_n$ | $n \geq 0$, $w_i \in \{0,1\}^*$, $\#1(w_i) >= \#1(w_{i+1})$, $\#1(w_i) <= 3$, where #1(w) counts the number of "1" belonging to w}

For instance, vald stings of L are: 0, 01010100, 111\$00010010, 111\$100\$1\$0000001\$0

1. Write the minimum power automaton recognizing the language.
2. Write a grammar (not necessarily minimal) that generates the language L.
3. Say what kind of automaton is required to recognize the language L' defined as follows:
   L' = { $w_1\$w_2\$\ldots\$w_n$ | $n \geq 0$, $w_i \in \{0,1\}^*$, $\#1(w_i) >= \#1(w_{i+1}$, where #1(w) counts the number of "1" belonging to w}
   (L' is defined as L without the constraint on the number of '1' that can appear in w_i.)

Solution to point 1

The language can be recognized by an FSA like the one shown in Figura 1-107.

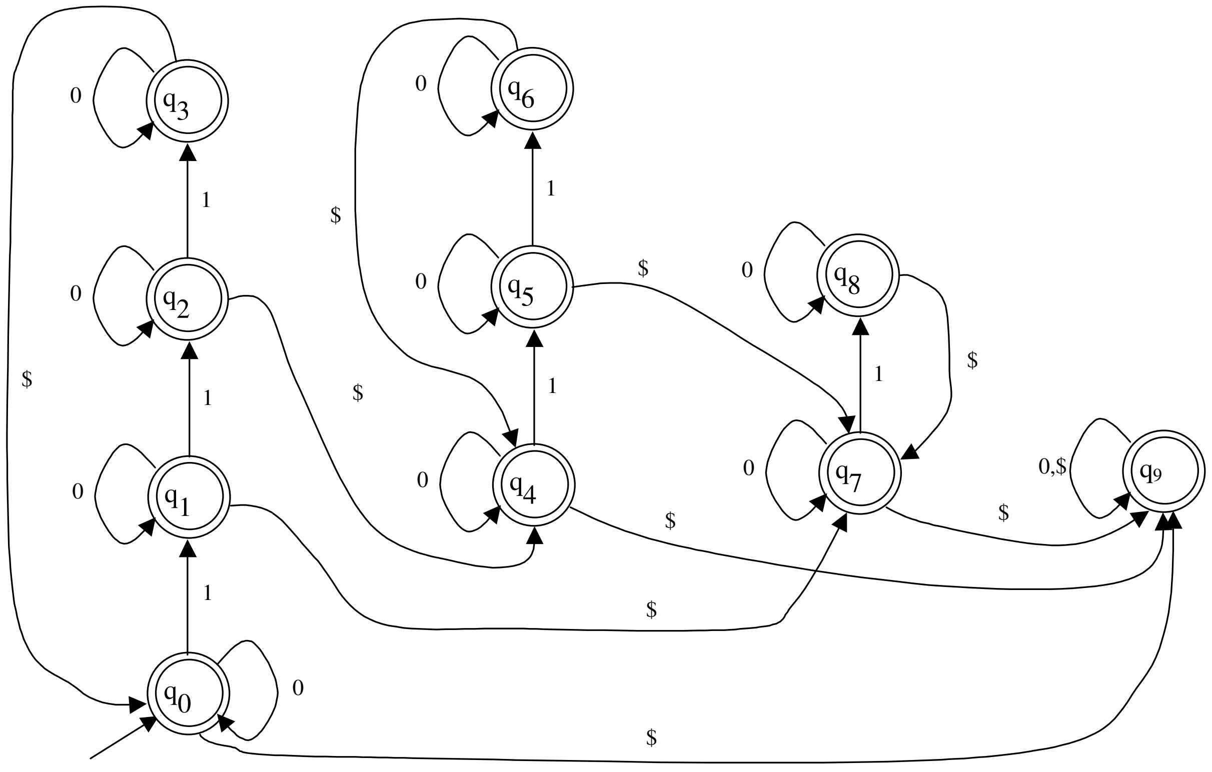


Figura 1-107 FSA recognizing L = { $w_1\$w_2\$\ldots\$w_n$| $n\geq0$, $w_i\in\{0,1\}^*$, $\#1(w_i) \geq \#1(w_{i+1})$, $\#1(w_i) \leq 3$.

Solution to point 2

A grammar generating the language L is the following:

$S \rightarrow S_3$

$S_3 \rightarrow W_3\, R_3 \mid S_2 \mid S_1 \mid S_0 \mid \varepsilon$

$R_3 \rightarrow \$S_3 \mid \varepsilon$

$S_2 \rightarrow W_2\, R_2 \mid S_1 \mid S_0 \mid \varepsilon$

$R_2 \rightarrow \$S_2 \mid \varepsilon$

$S_1 \rightarrow W_1\, R_1 \mid S_0 \mid \varepsilon$

$R_1 \rightarrow \$S_1 \mid \varepsilon$

$S_0 \rightarrow W_0 R_0 \mid \varepsilon$

$R_0 \rightarrow \$S_0 \mid \varepsilon$

$W_3 \rightarrow 0W_3 \mid 1\ W_2$

$W_2 \rightarrow 0W_2 \mid 1\ W_1$

$W_1 \rightarrow 0W_1 \mid 1\ W_0$

$W_0 \rightarrow 0W_0 \mid \varepsilon$

Of course it is possible to write a regular grammar, but it would be more complex.

Solution to point 3

The recognition of L' requires to count $\#1(w_i)$ and $\#1(w_{i+1})$, to compare them and to keep track of $\#1(w_{i+1})$ for subsequent comparison with $\#1(w_{i+2})$ etc…

A FSA is not suitable for this purpose since $\#1(w_i)$ is unlimited. It is not possible to use a PDA because after the comparison of $\#1(w_i)$ and $\#1(w_{i+1})$ this number must still be available. It is therefore necessary an automaton equivalent to a TM.

Exercise 1.7.9

Build a NPDA for the following laungage

$L = \{a^n b^n \mid n \geq 1\} \cup \{ a^n b^{2n} \mid n \geq 1\}$.

Solution

A NPDA that recognizes L is reported in Figura 1-108.

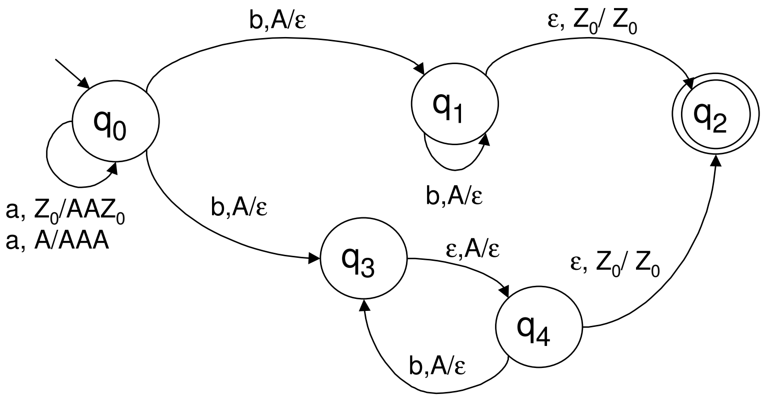


Figura 1-108 A NPDA recognizing $L = \{a^n b^n \mid n \geq 1\} \cup \{ a^n b^{2n} \mid n \geq 1\}$.

Exercise 1.7.10

Write the minimal power automaton recognizing the language $L = \{ x \in \{a,b,c\}^* \mid \#a(x)=\#b(x) \text{ or } \#a(x)=\#c(x) \}$

where, for a character *a* and a string *x*, the expression *#a(x)* denotes the numer of occurrences of *a* in *x*.

Solution

Consider that $L = L_1 \cup L_2$, where

$L_1 = \{ x \in \{a,b,c\}^* \mid \#a(x)=\#b(x)\}$

$L_2 = \{ x \in \{a,b,c\}^* \mid \#a(x)=\#c(x) \}$

The language can be recognized by a NPDA builded as the union of the DPDA that recognize L_1 and L_2, as shown in Figura 1-109.

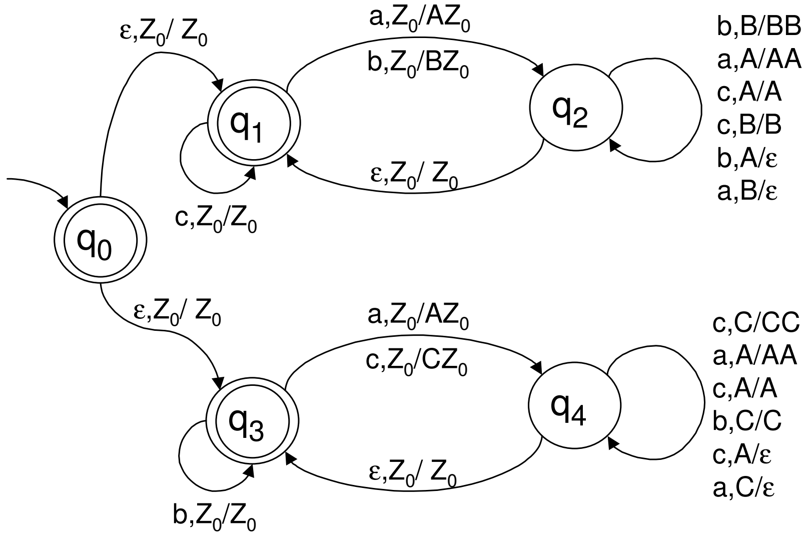


Figura 1-109 An NPDA recognizing $L = \{ x \in \{a,b,c\}^* \mid \#a(x)=\#b(x) \text{ or } \#a(x)=\#c(x) \}$.

The demonstration that the language L is not regular can rely on the pumping lemma for context-free languages:

x = abc

x = uywzv

y = a

w = b

z = c

u = v = ε

$\forall\, i \geq 0$, uy^iwz^iv belongs to language. In fact b, abc, aabcc, aaabccc, aaaabcccc, ecc. belong to language.

Exercise 1.7.11

Build a grammar generating the language of all permutations of strings like $a^nb^nc^nd^n \mid n \geq 1$.

Solution

A simple solution, obtained by writing explicitly the permutation rules, is the following:

$S \rightarrow ABCDS \mid ABCD$

$AB \rightarrow BA$

$BA \rightarrow AB$

$AC \rightarrow CA$

$CA \rightarrow AC$

…

$A \rightarrow a$

$B \rightarrow b$
$C \rightarrow c$
$D \rightarrow d$

Exercise 1.7.12 (*)

Build a grammar G generating the following language:

$L = \{(a^n b^n)^m \mid n \geq 1 \text{ e } m \geq 1\}$

(Notice that the value of n must be unique within the same string: for instance, the string aabbaaabbbaabb does not belong to L)

Solution

The following are the main steps to generate the grammar:

1. With the following *macro-rules*
   $S \rightarrow XA^+KD$,
   $D \rightarrow B^+HE$,
   $E \rightarrow A^+KD \mid Y$
   a first derivation is obtained.
   $S \Rightarrow^* XA^{n1}KB^{n2}HA^{n3} \ldots B^{nk}HY$
2. Then, using the following rules:
   $XA \rightarrow AXN$,
   $NA \rightarrow AN$
   $XK \rightarrow K$
   we get to:
   $\Rightarrow^* A^{n1}N^{n1}KB^{n2}\, H\ldots.HY$ (in case that some A are not used, the derivation process is blocked)
3. $NKB \rightarrow BKM$,
   $MB \rightarrow BM$,
   $NB \rightarrow BN$,
   in this way "N goes through K", bringing every B on the left of K and counting them based on n_1. So we get:
   $\Rightarrow^* A^{n1}B^{n1}K\, M^{n1}H \ldots.HY$ (if n1 = n2)
   $\Rightarrow^* A^{n1}\, B^{n2}\, N^{n1-n2}\, K\, M^{n2}H \ldots.HY$ (if n1 > n2)
   $\Rightarrow^* A^{n1}\, B^{n1}K\, M^{n1}B^{n2-n1}H \ldots.HY$ (if n2 > n1)
4. $MHA \rightarrow HAN$,
   $NA \rightarrow AN$,
   $NA \rightarrow AN$,
   $A \rightarrow a$
   $B \rightarrow b$
   M goes through H, leaving on the right of H the same number of As and Ns. Then the terminal characters a and b are produced.
   $\Rightarrow^* a^{n1}b^{n1}KH\, a^{n1}\, N^{n1}K \ldots.HY$ (se n1 = n2 = n3)
5. $KH \rightarrow \varepsilon$
   $KHY \rightarrow \varepsilon$

If not every n_i are the same, some N or M remains unbalanced between terminals H and K, but K and H are deleted only if they are adjacent.

Exercise 1.7.13

Build a grammar, possibly regular, generating the language L, defined on the alphabet {a,b,0,1}, in which the strings *x* meets the following requirements:

- if x starts with a, then it contains an even number of 1 and an odd number of 0;
- if x starts with b, then it contains an odd number of 1 and a even number of 0.

Solution

First of all, we can notice that, since nothing has been said about the strings not starting with a or b, they belong to the language.

It is possible to build an automata recognizing L, then an equivalent regular grammar by using the following states:

- q_1, q_5: an even number of 0 and an even number of 1 has been read.
- q_2, q_8: an even number of 0 and an odd number of 1 has been read.
- q_3, q_7: an odd number of 0 and an odd number of 1 has been read.
- q_4, q_6: an odd number of 0 and an even number of 1 has been read.

Such FSA is reported in Figure 1-110.

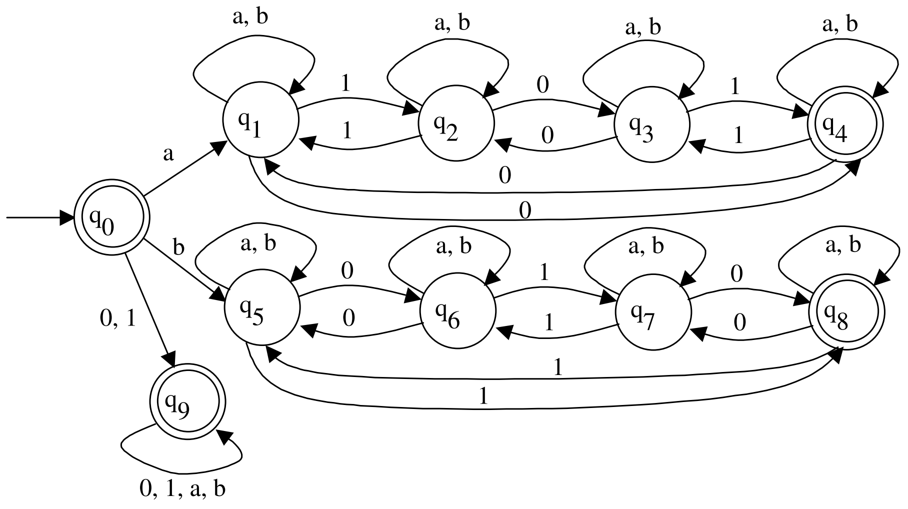


Figure 1-110

The corresponding grammar is the following:

$S \to a\ S_1 \mid bS_5 \mid 0\ S_9 \mid 1\ S_9 \mid \varepsilon$

$S_1 \to a\ S_1 \mid bS_1 \mid 1S_2 \mid 0\ S_4$

$S_2 \to a\ S_2 \mid bS_2 \mid 1\ S_1 \mid 0\ S_3$

$S_3 \to a\ S_3 \mid b\ S_3 \mid 0\ S_2 \mid 1\ S_4$

$S_4 \to a\ S_4 \mid b\ S_4 \mid 0\ S_1 \mid 1\ S_3 \mid \varepsilon$

$S_5 \to a\ S_5 \mid bS_5 \mid 0S_6 \mid 1\ S_8$

$S_6 \to a\ S_6 \mid bS_6 \mid 0\ S_5 \mid 1\ S_7$

$S_7 \to a\ S_7 \mid bS_7 \mid 0\ S_8 \mid 1\ S_6$

$S_8 \to a\ S_8 \mid bS_8 \mid 0\ S_7 \mid 1\ S_5 \mid \varepsilon$

$S_9 \rightarrow a\ S_9 \mid b\ S_9 \mid 0\ S_9 \mid 1\ S_9 \mid \varepsilon$

Exercise 1.7.14

Let k be a fixed positive integer (for instance 12, 333, 110002, …). Build a grammar generating the language $L = L_1 \cup L_2 \cup L_3 \ldots \cup L_k$ with $L_i = \{a^n b^{in} \mid n > 0\}$.

Solution

$S \rightarrow S_1 \mid S_2 \mid S_3 \mid \ldots \mid S_k$

$S_1 \rightarrow aS_1b \mid ab$

$S_2 \rightarrow aS_2bb \mid abb$

$S_3 \rightarrow aS_3bbb \mid abbb$

e $\forall\ i \leq k$

$S_i \rightarrow aS_ib^i \mid ab^i$

Exercise 1.7.15

Build a grammar generating the language $L = \{a^n b^{in} \mid n > 0, i > 0\}$.

Solution

The grammar generating L has to first generate a string XA^nPY

$S \rightarrow XPY$

$X \rightarrow XA$

Then P will be moved through the A, with rules like $AP \rightarrow PAB$: for each A crossed it will be produced a B. The result is $XP(AB)^iY$.

B itself will be permuted with the A ($BA \rightarrow AB$) until it reaches Y. the result is XPA^nB^nY.

At that point it has to be transformed into *b* ($BY \rightarrow Yb$).

The result, so far, is: XPA^nYb^n.

When P is near to X it is possible to choose between $XP \rightarrow XQ \mid XP \rightarrow \$$

Q "comes back" to Y with the rule $QA \rightarrow AQ$. Once it reaches Y, it is transformed back into P ($QY \rightarrow PY$) and it is ready for a new iteration.

Finally \$ goes through the A for the last time turning them in *a* ($\$A \rightarrow a\$$, $\$Y \rightarrow \varepsilon$).

The following is an exaple of the derivation of aabbbbbb (n=2, i=3)

$S \Rightarrow XPY \Rightarrow XAPY \Rightarrow XAAPY \Rightarrow XAPABY \Rightarrow XPABABY \Rightarrow XPAABBY \Rightarrow XPAABYb \Rightarrow XPAAYbb \Rightarrow XQAAYbb \Rightarrow XAQAYbb \Rightarrow XAAQYbb \Rightarrow XAAPYbb \Rightarrow XAPABYbb \Rightarrow XPABABYbb \Rightarrow XPAABBYbb \Rightarrow XPAABYbbb \Rightarrow XPAAYbbbb \Rightarrow XQAAYbbbb \Rightarrow XAQAYbbbb \Rightarrow XAAQYbbbb \Rightarrow XAAPYbbbb \Rightarrow XAPABYbbbb \Rightarrow XPABABYbbbb \Rightarrow XPAABBYbbbb \Rightarrow XPAABYbbbbb \Rightarrow XPAAYbbbbbb \Rightarrow \$AAYbbbbbb \Rightarrow a\$Aybbbbbb \Rightarrow aa\$Ybbbbbb \Rightarrow aabbbbbb$

Exercise 1.7.16

a. Build a grammar that generates the following language:
   $L = \{a^{n_1}ba^{n_2}ba^{n_3}b\ldots a^{n_k}c^m \mid n \geq 0, k \geq 0, n_1+n_2+\ldots n_k = m\}$

b. Say, motivating the answer, what automatons among the followings, are able to recognize L:
deterministic finite state automatons,
nondeterministic finite state automatons,
deterministic pushdown automatons,
nondeterministic pushdown automatons,
deterministic Turing machines,
nondeterministic Turing machines.

Solution to point a

$S \rightarrow aSc \mid bS \mid \varepsilon$

Solution to point b

The grammar is context-free, so the language is recognized by a nondeterministic pushdown automaton and automatons in more powerful classes (Turing machines).

In fact it is possible to mechanically build a nondeterministic pushdown automaton equivalent to the given grammar. That NPDA is shown in Figure 1-111.

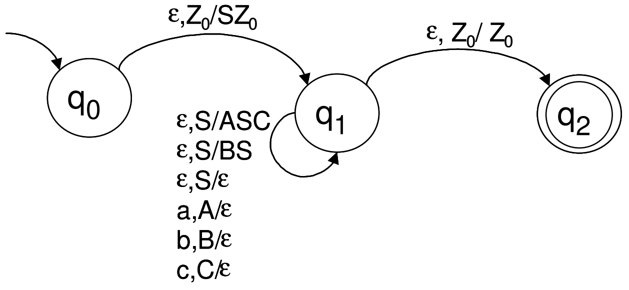


Figure 1-111 A NPDA recognizing $L = \{a^{n_1}ba^{n_2}ba^{n_3}b...a^{n_k}c^m | n_i \geq 0, k \geq 0, n_1+n_2+...n_k = m\}$.

Actually this particular grammar is recognized by a DPDA that initially push the a and ignores the b, when it reads the first c it goes in a state in which pops an A for each c read. Accepted strings is where the number of a and c is equal.

Such an automaton is shown in Figure 1-112.

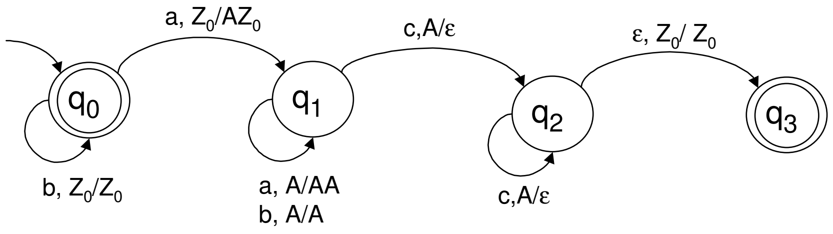


Figure 1-112 A DPDA recognizing $L = \{a^{n_1}ba^{n_2}ba^{n_3}b...a^{n_k}c^m | n_i \geq 0, k \geq 0, n_1+n_2+...n_k = m\}$.

Exercise 1.7.17

2. Si dica se l'automa ottenuto nella parte 1 è un automa a pila deterministico o no. Nel caso negativo si dica se è possibile ottenerne uno equivalente deterministico ed in tal caso costruirlo.

Build an automa ton recognizing the language generated by the following grammar:

S → AB
A → aAa | ε
B → bBc | bc

Say whether the automaton obtained is a deterministic PDA or not. If it is not deterministic say if it is possible to obtain an equivalent deterministic PDA and build it.

Solution to point 1

It is possible to obtain an NPDA starting from the context-free grammar. To make it easier it is better to use an equivalent grammar:

S → AB
A → CAC | ε
C → a
B → KBD | KD
K → b
D → c

The corresponding automaton is reported in Figure 1-113.

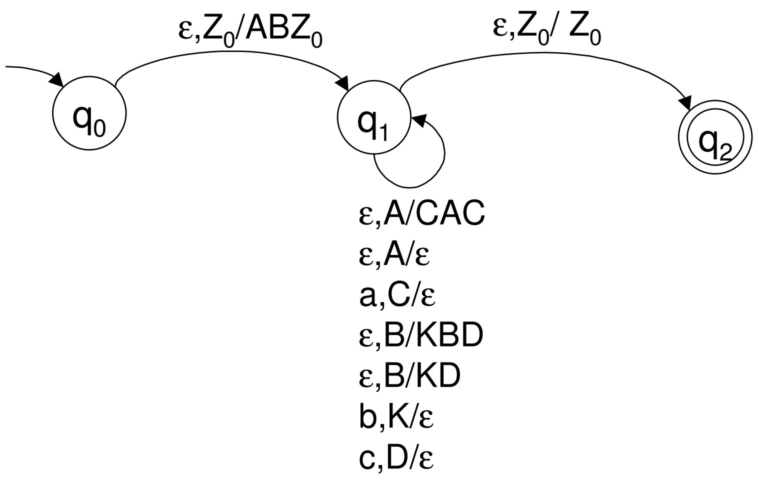


Figure 1-113

Solution to point 2

The automaton is clearly nondeterministic. A deterministic PDA is obtainable by observing that the language generated by the given grammar is $L = \{a^{2n}b^mc^m \mid n \geq 0, m > 0\}$ and building the corresponding DPDA. Such an automaton is reported in Figure 1-114.

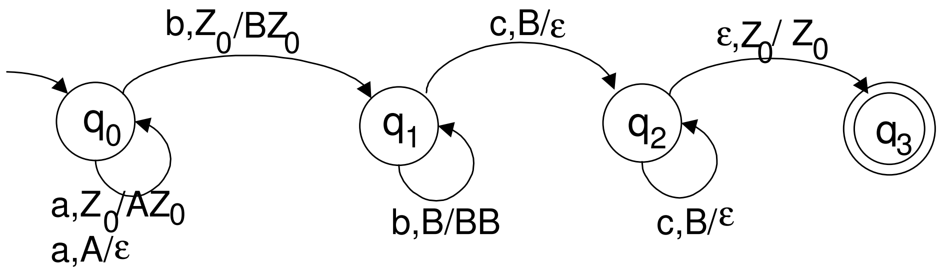


Figure 1-114 DPDA recognizing $L = \{a^{2n}b^mc^m \mid n \geq 0, m > 0\}$.

Exercise 1.7.18

Consider the following languages:

$L_1 = \{ x \in \{a,b,c\}^* \mid \#a(x) = \#b(x) \text{ or } \#a(x) = \#c(x) \}$

$L_2 = \{ x \in \{a,b,c\}^* \mid \#b(x) = \#a(x) - \#c(x) \text{ e } \#a(x) > \#c(x)\}$

where, for a character α and a string x, the expression $\#\alpha(x)$ denotes the number of times the character α is repeated in the string x. e.g. L_1 includes the strings ab", "ac", "acacacacaaaccc", "abc", "abcac", but not "abac" e "aba". L_2 includes the following strings: "bcaa", "abacaabb", but not "ac", "aaabc", "aabca".

a) what kind of automaton is needed to recognize L_1?

b) what kind of automaton is needed to recognize L_2?

c) what kind of automaton is needed to recognize the intersection of the two languages?

Explain your answers.

Solution to point a)

NDPDA (it is sufficient to compose in a nondeterministic way the DPDA recognizing $\{x \in \{a,b,c\}^* \mid \#a(x) = \#b(x)\}$ with the DPDA recognizing $\{x \in \{a,b,c\}^* \mid \#a(x) = \#c(x)\}$.

Solution to point b)

DPDA: $L_2 = \{ x \in \{a,b,c\}^* \mid \#b(x) = \#a(x) - \#c(x) \text{ and } \#a(x) > \#c(x)\} = \{ x \in \{a,b,c\}^* \mid \#b(x) + \#c(x) = \#a(x) \text{ and } \#a(x) > \#c(x)\}$

When a *b* or *c* is read, a symbol is pushed on the stack. The same symbol is used to represent both the carachters. If *#b(x) + #c(x) = #a(x)*, the condition *#a(x) > #c(x)* is guaranteed if *#b(x)>0*, therefore the automaton has to stay on a non-recognizing state until it reads a b.

The language L_2 is recognized by the DPDA in Figure 1-115.

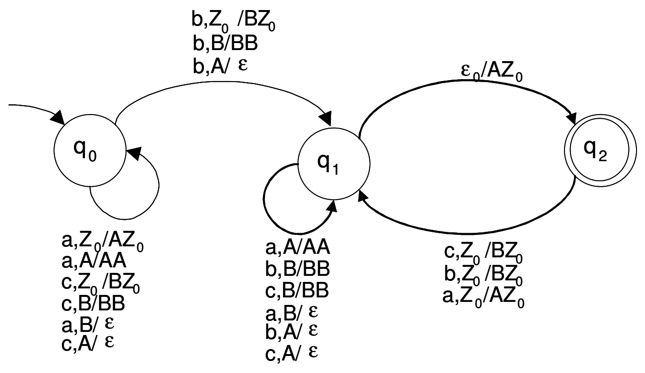


Figure 1-115

Solution to point c)

$L = L_1 \cap L_2 = \{ x \in \{a,b\}^* \mid \#a(x) = \#b(x)\}$

The language L is recognized by a DPDA.

Exercise 1.7.19

Build a grammar that generates language L, defined as follows:

$\forall x\ (x \in L \leftrightarrow \exists y,z,w\ (x = y.z.w \wedge y \in L_1 \wedge z \in L_2 \wedge w \in L_3)$

$\wedge$

$\forall x\ (x \in L_1 \leftrightarrow (x = aa \vee x = bb \vee \exists y\ (x = a.y.a \wedge y \in L_1 \vee x = b.y.b \wedge y \in L_1)$

$\wedge$

$\forall x\ (x \in L_2 \leftrightarrow (x = \varepsilon \vee \exists y\ (x = ab.y \wedge y \in L_2)$

$\wedge$

$\forall x\ (x \in L_3 \leftrightarrow (x = \varepsilon \vee \exists y\ (x = a.y.b \wedge y \in L_3)$

Solution

L is obtained concatenating L_1, L_2, L_3:

$L_1 = \{x \mid x = w.w^R, w \in \{a,b\}^+\}$, $L_2 = \{ab\}^*$ e $L_3 = \{a^n b^n, n >= 0\}$

Hence, the following grammar generates L:

$S \rightarrow ABC$

$A \rightarrow aa \mid bb \mid aAa \mid bAb$

$B \rightarrow \varepsilon \mid abB$

$C \rightarrow \varepsilon \mid aCb$

Exercise 1.7.20

Consider the language L defined as follows:

$s \in L \leftrightarrow \exists\ y,m\ (s=c^m.y \wedge m>0 \wedge y \in L_Y)$

$s \in L_Y \leftrightarrow s=\varepsilon \vee \exists\ x,m\ (s=a^m.x \wedge m>0 \wedge x \in L_X)$

$s \in L_X \leftrightarrow s=\varepsilon \vee \exists\ y,m\ (s=b^m.y \wedge m>0 \wedge y \in L_Y)$

1. Write a grammar generating L.
2. Say, justifying your answer, which is the minimal power automaton recognizing L.

Solution to point 1.

L is defined as follows: L= $\{x \mid x = c^m.(a^k.\ y)^k,\ m>0,\ k\in\{0,1\}$ e $y\in\{a,b\}^*\}$

A grammar generating L is the following:

$S \rightarrow cS \mid cY$

$Y \rightarrow \varepsilon \mid AX$

$X \rightarrow \varepsilon \mid BY$

$A \rightarrow a \mid aA$

$B \rightarrow b \mid bB$

Solution to point 2.

Since there is no correlation between the number of times that a, b, c appear in L, the minimal power automaton that recognizes the language is a finite state automaton.

An automaton recognizing L is shown in Figure 1-116.

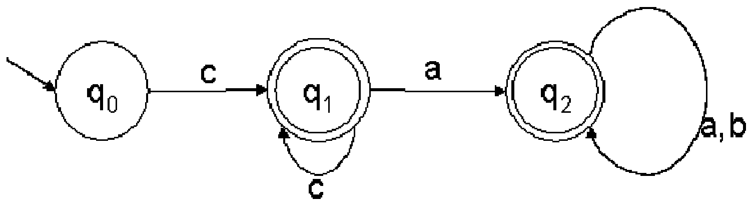


Figure 1-116 Finite state automaton recognizing L= $\{x \mid x = c^m.(a^k.\ y)^k,\ m>0,\ k\in\{0,1\}$ e $y\in\{a,b\}^*\}$.

Exercise 1.7.21

1. Formalize through an appropriate abstract machine the behaviour of a control system designed to monitor a plant. When the system working fine, it receives a signal from a sensor installed on the plant at regularly spaced instants (e.g. with a period of k seconds).
   If the signal does not show any anomaly the system remains in normal operating conditions. If the signal indicates the presence of an anomaly, the control system can either decide to turn off the system or to report it to a human operator. In the first case the system remains off until it is given a command to restart from the outside. In the second case, if the operator performs an action to repair the system, it resumes normal operation, but if the operator takes no action within *k* seconds, or he sends an explicit command to shutdown, the system is turned off and remains off until is not given a command to restart from the outside.
2. Say if the abstract machine built in step *a* of the exercise is deterministic or not. In the second case, is it possible to build an equivalent deterministic machine? In case of positive answer, built it.
3. Formalize through a first order formula, the following property: "the system is never in abnormal operating conditions more than x seconds".

Solution to point 1.

A suitable abstract machine describing the behaviour of the system is the non-deterministic finite automaton shown in Figure 1-117.

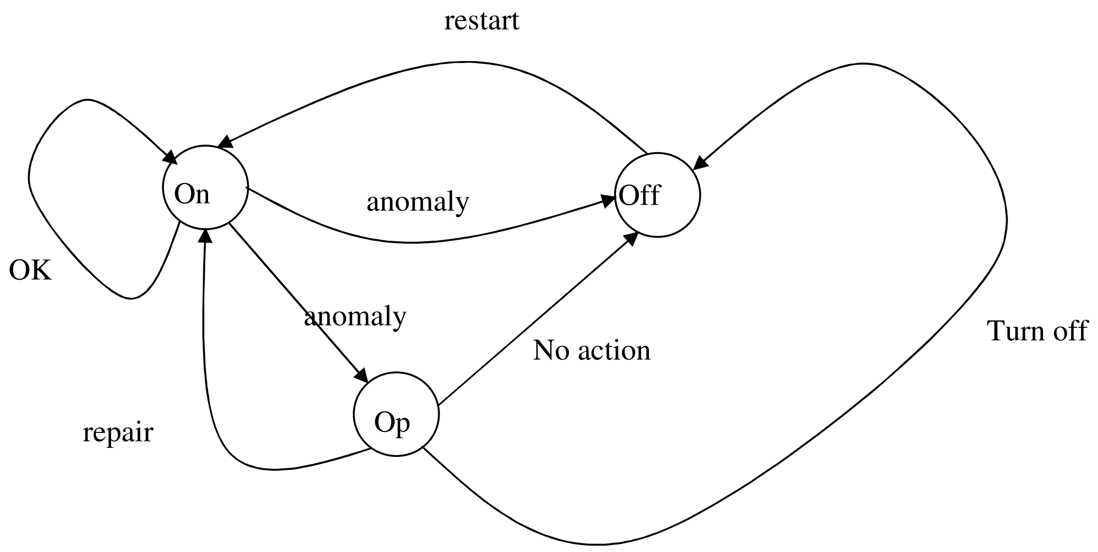


Figure 1-117 A non-deterministic FSM representing the behaviour of a monitoring system.

Solution to point 2.

The "natural" finite state automaton describing the operation of the system is nondeterministic, as shown in the solution of step 1. An equivalent deterministic version is shown in Figure 1-118.

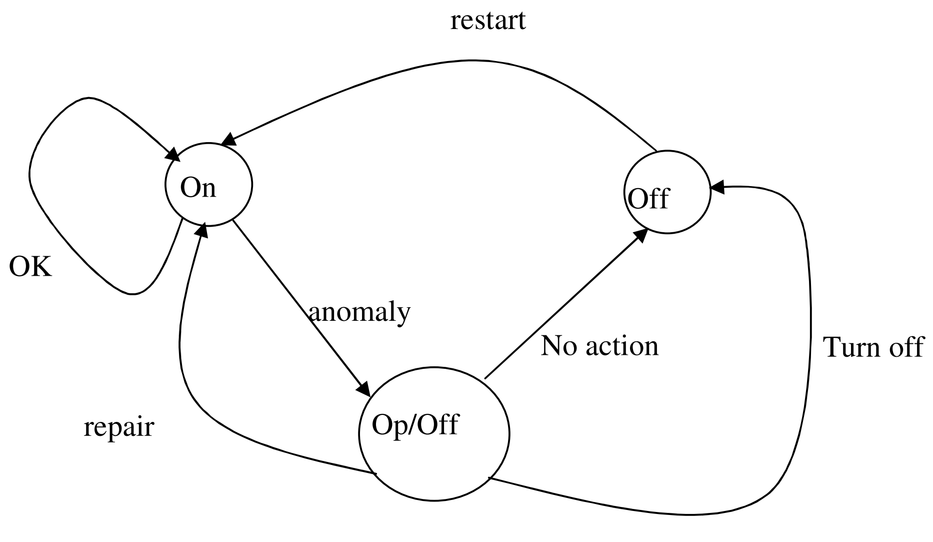


Figure 1-118 A deterministic FSM representing the behaviour of a monitoring system.

Solution to point 3

Use the symbol *Nor(t)* to state that the system is in normal operating condition at time t. The following formula represents the required property:

$\forall h (\forall t (0 \leq t \leq h) \rightarrow \neg Nor(t)) \rightarrow h < x$

Exercise 1.7.22

1. Formally define the acceptance condition of a string by a pushdown automaton recognizing from empty stack and final state. The string is accepted if and only if, after scanning it, the stack of the automaton is empty and the state of the controller belongs to the set of the acceptance states.
2. Say, briefly justifying the answer, if the nondeterministic pushdown automata that recognize empty stack and final state are more, less or equally powerful to the nondeterministic automata that recognize only on the basis of the final state.
3. Say, briefly justifying the answer, if the nondeterministic pushdown automata that recognize empty stack and end state are more, less or equally powerful to the nondeterministic automata that recognize only from empty stack (ie accepting the input string if and only if at the end of its scanning the stack is empty).

Solution to point 1.

Adopting the usual conventions, x is accepted if and only if
$c_0 = <q_0, x, Z_0> |\text{---}^* <q, \varepsilon, \varepsilon> \wedge q \in F$

Solution to point 2.

The new family of automata is as powerful as the one recognizing only when the automata is in a final state. In fact, given an automaton that recognizes only the final state, it can be transformed into an automaton behaving exactly like the original, but once arrived in a state of acceptance of the original (which will no longer be accepted for the new automaton), can bring in a new state of acceptance through an ε-move and then empties the stack.

Note however that such a transformation would transform an originally deterministic automaton in a nondeterministic one, since the states of acceptance does not necessarily lead to the halt of the automaton.

On the other hand, given an automaton recognizing when it is in a final state and its stack is empty, it can be transformed into an automaton recognizing when it is in a final state. At first there is the need to transform it into an automaton that uses the symbol Z_0 to represent only the "bottom of the stack" and that then deletes it only in the last move. This can always be done possibly by introducing a new symbol Z_{00} if Z_0 already did not hold this property. Then every move that deletes the "bottom of the stack" and goes into an acceptance state has to be transformed into a move that just brings the automaton in a unique acceptance state in which it is blocked.

Solution to point 3.

La nuova famiglia di automi è ugualmente potente anche di quella che riconosce a sola pila vuota. Infatti, mediante costruzioni simili alle precedenti, è sempre possibile costruire un automa che riconosca a stato finale modificando uno che riconosca a pila vuota e viceversa.

The new family of automata is as powerful as the one recognizing only at empty stack. In fact, using constructions similar to the ones of the previous point, it is always possible to build an automaton that recognizes at final state modifying the automaton which recognizes at empty stack and vice versa.

Exercise 1.7.23

Let $\tau(x)$, $x \in \{a,b,c\}^*$ be the translation defined by the pushdown translator in Figure 1-119. Note that, in the convention used in the figure, the rightmost character is placed on the top of the stack while the one on the left stays below it.

Build a grammar generating the language defined as follows: $L = \{x\$\tau(x), x \in \{a,b,c\}^*\}$.

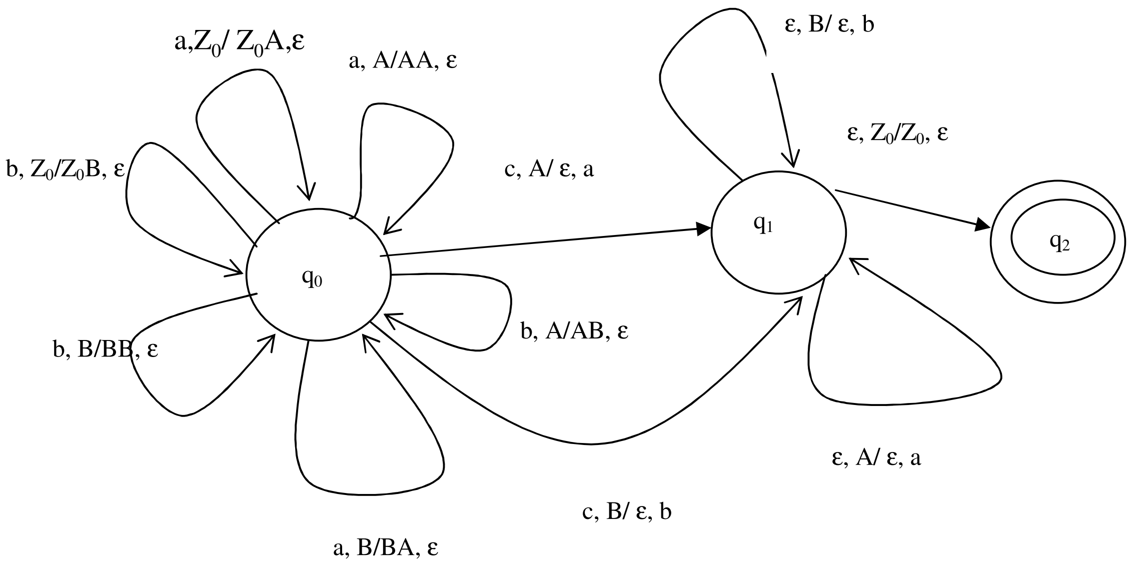


Figure 1-119 Pushdown translator

Solution

The translator in Figure 1-119 takes as input a string $x=y.c$ with $y\in\{a,b\}^*$ and translates it into $\tau(x)=y^R$.

A grammar generating L is the following:
$S \rightarrow aAa \mid bAb$
$A \rightarrow aAa \mid bAb \mid c\$$

Exercise 1.7.24

A multi-pass compiler reads the source file from the input and produces the object code in an output file after the execution of several steps, each one producing files containing intermediate representations: the input of each step is the output of the previous.

1. Formalize the concept of *k-pass deterministic pushdown automaton*. In analogy with the multi-pass compiler, it receives a string on the input tape and translates it like a regular deterministic pushdown automaton. When it is done it repeates the operations on the produced string for k times.

2. Say, briefly justifying the answer, whether the k-steps automaton used as a recognizer of languages (it performs k-1 translation and at the k-th step it operates only as a recognizer accepting the original string if and only if the string produced by the (k-1)-th step is accepted by the k-th step) is more powerful than the usual deterministic pushdown automaton.

Solution to point 1.

The elements of the automaton (state set, alphabet, transitions, ...) are identical to those of normal deterministic translator, the only difference is that the alphabets I and O have to be the same.

The acceptance and the translation of a string by the automaton with more passes are defined as follows:

$x \in L(T) \wedge \tau(x) = z \leftrightarrow <x, q_0, Z_0, \varepsilon> \stackrel{*}{\vdash\!\!-} <\varepsilon, q, \gamma, z_1> \wedge q \in F \wedge$

$< z_1, q_0, Z_0, \varepsilon> \stackrel{*}{\vdash\!\!-} <\varepsilon, q, \gamma, z_2> \wedge q \in F \wedge$

...

$< z_{k-1}, q_0, Z_0, \varepsilon> \stackrel{*}{\vdash\!\!-} <\varepsilon, q, \gamma, z_k> \wedge q \in F \wedge z_k = z$

The symbols q and γ used in the definition does not necessarily denote the same value.

Solution to point 2.

k-passes automata are more powerful than one-pass automata. It is easy to recognize the language {anbncn} in two steps: the first pass is to verify that the number of a is equal to the number of b and it copies b and c on the output. The second pass verifies that the number of b is equal to the number of c.

Exercise 1.7.25

Build a push-down automata recognizing the strings generated by the following grammar

<Exp> → x | (<Exp> + <Exp>) | (<Exp> - <Exp>) | (<Exp> * <Exp>) | <Cost>*<Exp>

<Cost> → 0 | 1 | 2 | … 9

Solution

A push-down automata recognizing the expressions generated by the given grammar is shown in Figure 1-120, where op is an abbreviation for the set of labels {+,-,*} and c is an abbreviation for the set the labels {0,1, …9}.

Exercise 1.7.26

Let A be the alphabet of the alphabetical lower-case characters (A={a,b,…,z}). The language L contains all the strings x of characters of A with the following properties:

- If x starts with character ‘a’, then it must contains a number of ‘b’ equals to the number of ‘z’;
- If x starts with the character ‘b’, then the sum of the number of ‘c’ and of the number of ‘h’ must be an even number.

For instance, the following strings belong to L: abbzzzb, ace, aaaaaaaaaaaaaaaazb, hgsdfahgsfdhaf, xxxxx, bch, bccch, bhh. On the contrary the following strings do not belong to L: abbzzb, aceb, bchc, bhbbbbb.

1. Build a grammar and an abstract machine with the minimum expressive power respectively generating and recognizing L.
2. Write a first order formula defining the language L. For the sake of simplicity, in the formula you can use the standard operators defined on strings and languages (concatenation, *, set operators, …) without giving an explicit formal description of them.

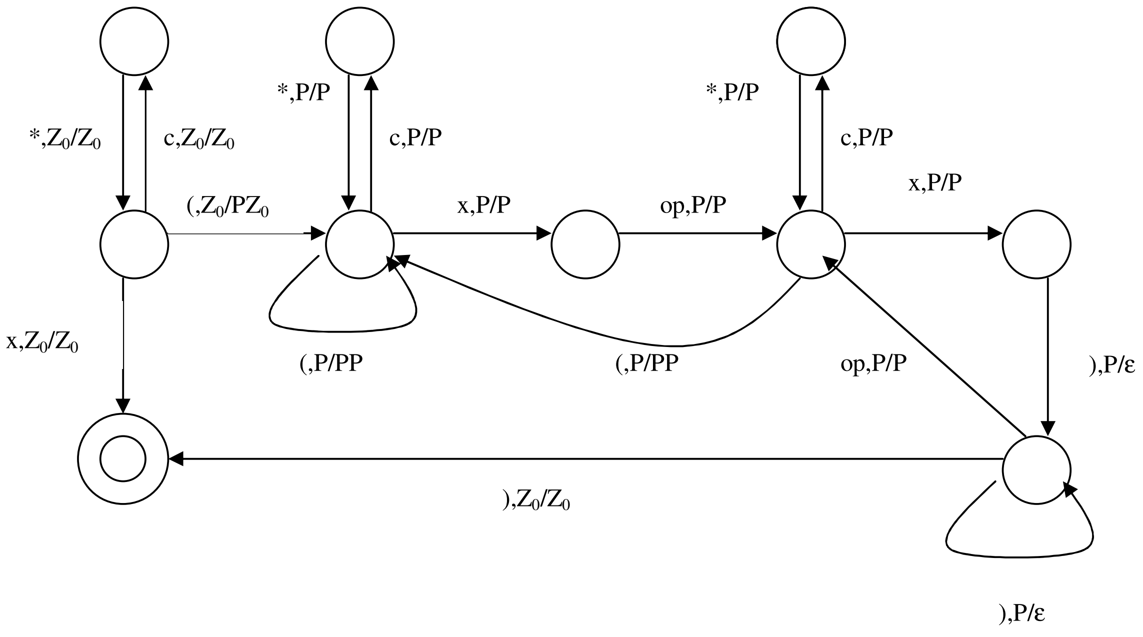


Figure 1-120 A PDA recognizing strings of arithmetic expressions.

Solution to step 1.

A context free grammar generating L is the following:

S → A | B | C

A → aH

H → bHzH | zHbH | ε | \$H (Where \$ represents any character of A different from ‘b’ and ‘z’)

B → bK

K → cJ | hJ | | ε | £K (Where £ represents any character of A different from ‘c’ and ‘h’)

J → cK | hK | | ε | £J

C → ε | &D (Where & represents any character of A different from ‘a’ and ‘b’)

D → ε | @D (Where @ represents any character of A)

A deterministic PDA can recognize L in the following way:

If the input string is empty or it does not start with 'a' or 'b', it is accepted in any case;

If the string starts with 'a' all characters different from 'b' and 'z' are ignored (using simple self-rings); the stack is used to count and remove the exceeding 'b' and 'z'

If the string starts with 'b' a simple FSA verifies if the number of 'c' and 'h' is even.

Solution to point 2.

A first order formula defining strings of L is the following:

$s \in L \leftrightarrow (\forall x$

$(s= a.x \rightarrow \#x_b = \#x_z) \wedge$

$(s= b.x \rightarrow \exists n (\#x_c + \#x_h = n \wedge \exists m (n = 2.m))$

$)$

Where operation # "number of occurrences of character a in x" is defined, as usual, in the following way (where b denotes any characters different from 'a'):

$x = \varepsilon \rightarrow \# x_a = 0 \wedge x = ay \rightarrow \# x_a = \# y_a + 1 \wedge x = by \rightarrow \# x_a = \# y_a$

Exercise 1.7.27

Tell, justifying the answer, if exist automatons belonging to the following families that recognize the language L defined as follows:
L = {a^n |, n numero primo > 12}. Families to teke in consideration are:

1) Deterministic finite state automatons
2) Nondeterministic finite state automatons
3) Deterministic Turing machines
4) Nondeterministic Turing Machines

Solution to point 1 and 2.
It not exists any DFSA or NFSA that recognizes L. Let's assume the opposite and use the pumping Lemma. Let *n* be a sufficient large prime (greater than the cardinality of Q): for the pumping lemma exists a *k*, with 1<=k<=n, that a^{n+rk} belongs to L, for all *r*. But, surely, for each *k* exists an *r* that n+rk is not prime.

Solution to point 3 and 4.
The problem of the primality of a number is notoriously decidable, therefore exist Turing machines, deterministic or not, that solve it.
Note: the fact that it has to be >12 is obviously irrelevant.

Exercise 1.7.28

Say, justifying the answer, which of the following sentences are true:

1) the complement of the intersection of two context-free languages is a context-free language
2) the complement of the intersection of two deterministic context-free languages is a context-free language
3) the complement of the intersection of two deterministic context-free languages is a deterministic context-free language
4) the complement of the intersection of two deterministic regular languages is a decidable language (Note: decidable languages are defined as decidable or recursive sets)

Solution to point 1.

The statement at point 1 is false. Indeed the context-free languages are not closed with respect to the complement: therefore exist a language L context-free such that it's complement L^ is not context-free. By intersecting L with the universal set, that is context-free, we still obtain L, whose complement L^ is not context-free.

Solution to point 2.

The statement at point 2 is true, in fact deterministic context-free languages are closed respect to the complement. Then the complement of the intersection of two deterministic context-free languages is the union of their complements, that are two deterministic context-free languages; the union of context-free languages is context-free.

Solution to point 3.

The statement at point 3 is false, in fact, as in the previous point, the complement of the intersection of two deterministic context-free languages is the union of two deterministic context-free languages, that may lead to a non-deterministic context-free language.

Solution to point 4.

The statement at point 4 is true. Thanks to the closure property of the regular languages respect to all set operations, the generated language is regular and so, obviously, decidable.

Exercise 1.7.29

Say, giving a justification of your answer, if the following statements are true or false:

1. The class of languages recognized by *deterministic* Turing machines that always halt their computation is closed under the complement.
2. The class of languages recognizable by *non-deterministic* Turing machines that always end their computation is closed under the complement.

Solution to point 1.

The statement in paragraph 1 is true. A TM recognizing a language L, always ending its computation, can always be transformed into a TM that recognizes L^ simply exchanging the states of acceptance with those of non-acceptance in the last move that involves the stop of the TM. NB: the class of languages recognized by *deterministic* Turing machines that always halts their computation is the same of the class of recursive languages.

Solution to point 2.

The statement in point 2 is true. A nondeterministic TM that always halts its computation can be transformed into a deterministic TM that has the same property: simply apply the standard construction of transformation from nondeterministic to deterministic and note that, if all the computations of the original machine always end, then, in the same way, the computations of the corresponding deterministic machine always end. Finally apply the previous property.

Exercise 1.7.30

Consider the language L defined over the alphabet {a, b, c, f} and consisting of all and only the strings belonging to {a, b, c}*.f containing a number of b equals to the number of c if they contain at least one a; containing, instead, a number of b equals to the number of c+10 if there are no a.

1. Describe, without necessarily going into details, an abstract machine that recognizes L. Prefer a machine with "minimal power" or choice between the family of automata less powerful among those able to recognize L.
2. Build a grammar G that generates L. Differently from the previous exercise, G is not necessarily of minimum power. Qualities of simplicity and naturalness (even if they are subjective) are desired for G in its construction and understanding.
3. Describe with a first-order formula all and only the strings belonging to the language L.

Solution to point 1.

The language L can be recognized by a deterministic pushdown automaton operating, in broad terms, as follows:

A "takes into account", through his stack, of the difference between the number of b and c (the stack contains Z_0 if and only if that difference is 0). If at least one a appears while reading, when f is read, the automaton verifies that in the stack there is Z_0; if it has not read any a, it uses the finite-state memory to verify that in the stack there is still an excess of exactly 10 b.

Solution to point 2.

There is certainly a contextual grammar that generates L, because L is recognized by a push-down automaton (deterministic). However, it is probably easier to generate L using the following grammar, even if it is of a more general type.

$S \rightarrow Hf$

$H \rightarrow AM \mid MBBBBBBBBBB$

$M \rightarrow BCM \mid AM \mid \varepsilon$

$AB \rightarrow BA$

$BA \rightarrow AB$

$CB \rightarrow BC$

$BC \rightarrow CB$

$AC \rightarrow CA$

$CA \rightarrow AC$

$A \rightarrow a$

$B \rightarrow b$

$C \rightarrow c$

Solution to point 3.

$x \in L \leftrightarrow x \in \{a,b,c\}^*.f \wedge$

$(((\exists y, z\ (x = y.a.z)\ \rightarrow \#(x,b) = \#(x,c)) \wedge ((\neg\ (\exists y, z\ (x = y.a.z))\ \rightarrow \#(x,b) = \#(x,c) + 10))))$

where the function #(x,b) represents the number of occurrences of character b in the string x, defined as following:

$$x = \varepsilon \rightarrow (\#(x,b) = 0) \wedge (x = by \rightarrow \#(x,b) = \#(y,b) + 1\ ) \wedge ((x = ay \vee x = cy) \rightarrow \#(x,b) = \#(y,b)\ )$$

The formula that formalizes the membership x ∈ {a,b,c}*.f is omitted, because it is well known

Exercise 1.7.31

1. Define roughly but with sufficient precision, a suitable abstract machine that models a device, informally defined as follows:

   A can receive from two different sensors a signal that can be the character 'a' or the character 'b'. The sensors can also not transmit any signal. If A receives the same signal from two sensors 'a' or 'b' whithin 2 seconds (the signals can also arrive at the same time), then after two seconds from the reception the second output signal, it emits the signal 'c'. After issuing a 'c', including the same moment when 'c' is emitted, A "loses memory" of previous events, and starts its operation as if it had not yet received any input signal. The issue of 'c' occurs only under the specified conditions.

   Assume that the time of A is discrete and that the abstract machine executes a transition synchronously every second.
2. Formalize, through an appropriate first-order formula, a new device A, removing the constraint of discreteness on time.

Solution to point 1.

The formalization of A through an abstract machine is conceptually simple but requires a considerable amount of detail. Given the features of A, the abstract machine can be a finite state automaton. In that case, its essential elements could be the following:

1. The input alphabet must describe the following conditions:
   a. no signal on both sensors
   b. signal *a* received by sensor 1 and no signal received by sensor 2
   c. signal *a* received by sensor 1 and signal *b* received by sensor 2
   d. signal *a* received by sensor 1 and signal *a* received by sensor 2
   e. …
2. The output alphabet formalizes the emission of *c* and the non-emission of any signal (plus any other signals not considered here)
3. A is initially in a state *q0* where it remains until it receives a signal from both sensors.
4. In any state (except in *qA0* and *qA1*, specified in this section) when A receives two equal signals from both sensors is carried in a waiting state *qA0*. From *qA0*, it goes to *qA1* after a unit of time and from *qA1* it has to go to *q0* emitting *c*.
5. When the device is in *q0* and it receives a signal on sensor 1 and no signal on sensor 2, it goes into a state in which it "counts" through appropriate transitions at successive time instants. If after two of such transitions it has not been received any signal, it goes back to *q0*; otherwise, if within the two transitions a signal is recived by sensor 2 the machine goes to *qA0* and it proceedes as above.

In any case, the different possible situations that have to be stored are finite, although they could be various and many.

NB: A more compact formulation could have been obtained with a Turing machine.

Solution to point 2.

Introduce the following predicates:

in_1(x, t): signal *x* is received at time *t* by sensor 1, $x \in \{a, b\}$

in_2(x, t): signal *x* is received at time *t* by sensor 2

out_c(t): signal *c* is emitted at time *t*.

The behavior of A is then described by the following formula:

$$\forall t(out\_c(t) \leftrightarrow (\exists t_1, t_2 (in\_1(x, t_1) \wedge (in\_2(y, t_2) \wedge |t_1 - t_2| \leq 2 \wedge (x = y) \wedge (t - \max(t_1, t_2) = 2)$$
$$\wedge$$
$$\forall t_3((\min(t_1, t_2) \leq t_3 \leq t) \rightarrow \neg out\_c(t_3))$$

Exercise 1.7.32

1. Define an appropriate abstract machine that models the system A defined informally as follows:

   A can receive from each sensor, at every second, a signal that can be the character 'a' or the character 'b'. But the sensor may not transmit any signal. If A receives the same signal from the sensor, both 'a' and 'b', within a range of k (> = 2) seconds, bouns included, then after two seconds of receiving the second signal, A emits the signal 'c'. Otherwise, A does not emit any signal. At any instant the condition for the emission of the output signal is recalculated regardless of what has been done before, ie c is emitted if and only if 2 seconds before it was verified the following condition: if in the range made by the k seconds preceding the current time, including the bounds, it has received a signal equal to the one of the current instant (NB: the term "the current time" refers to the time of the assessment of the condition, not at the time of any emission of 'c'): for example, for k = 3, if as input is received the sequence "aab-ab-a", the output string "- - - c - - cc - c" is emitted, where the symbol '-' indicates no signal.

   (NB: k is the length of the interval, that consists of k +1 instants, because the extremes of the range are included.)

   Assume that k = 2 in the description.

   Hint: A possible abstract machine might consist of two "automata in cascade", because the output alphabet of the first coinciding with the input alphabet of the second. The first automaton verifies at any time if the condition is required for the future emission of signal 'c'. If the condition is satisfied, it immediately sends an appropriate signal to the second one. The second automaton does nothing but emits the signal 'c' after two seconds.

2. Formalize the device A, described in the preceding paragraph, using an appropriate first-order formula.

Solution to step 1.

The pair of automata in Figure 1-93 and Figure 1-94 operates as shown in the hint. Several other solutions are possible.

In both figures the symbol s indicates that the signal is sent to the second automaton. No output stated implies the null string, that is precisely the lack of signal.

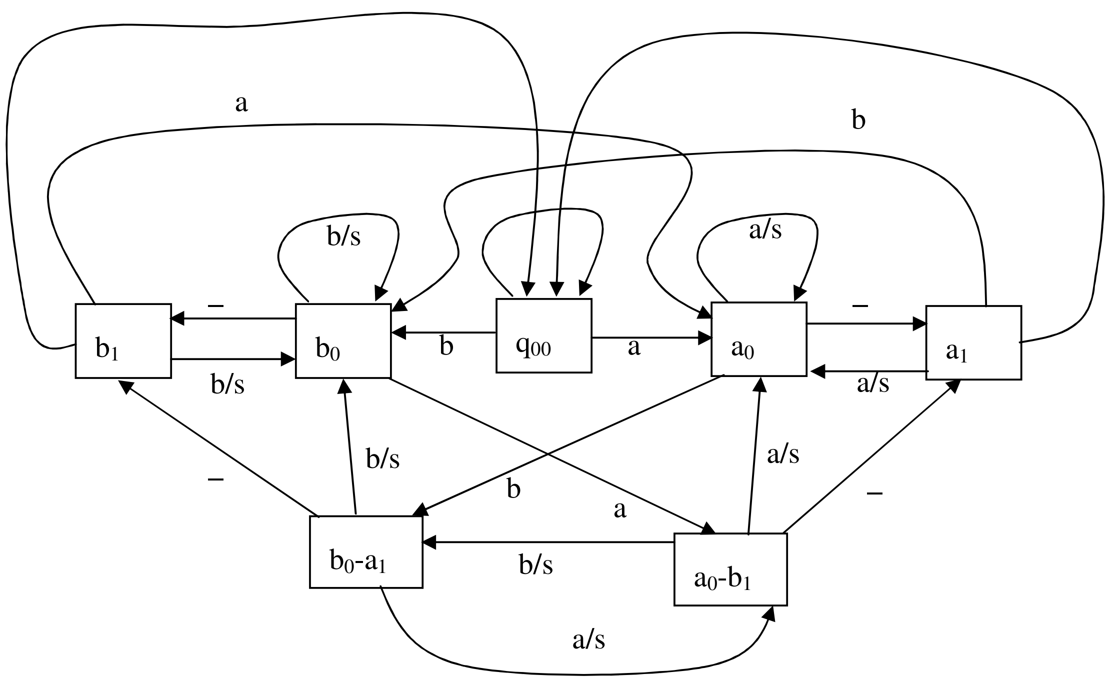


Figura 1-121

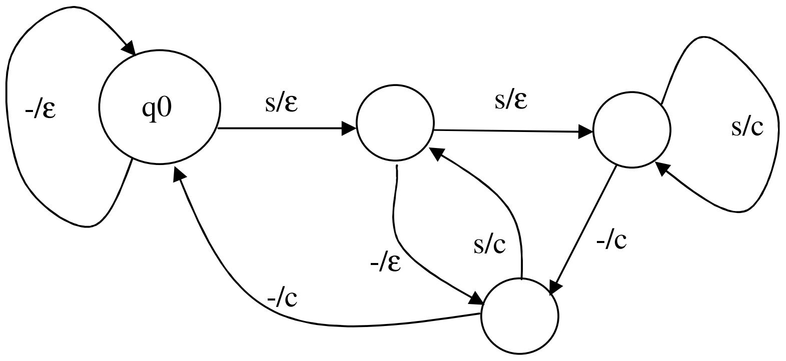


Figura 1-122

Solution to step 2.

We introduce the following predicates:

in(x, t): signal x is recived from sensor at time t, $x \in \{a, b, -\}$. '–' states no signal.

out_c(t): signal c is emitted at time t.

The behaviour of A is then desrcibed by the following formula:

$$\forall t(out\_c(t) \leftrightarrow$$
$$(\exists t_1, t_2 (in(x,t_1) \wedge in(y,t_2) \wedge |t_1 - t_2| \leq k \wedge (x = y) \wedge (x \neq -) \wedge (t_1 \neq t_2) \wedge (t - \max(t_1,t_2) = 2)))$$
$$)$$

Exercise 1.7.33

Consider the language L defined in the alphabet {a, b, c}, consisting of all and only the strings belonging to {a, b, c} * containing at least one b and ending with five consecutive c:

1. Describe an abstract machine recognizing L. Prefer a machine of "minimal power" choosing the family of automata less powerful among those able to recognize L.
2. Write a grammar G generating L. In this case, prefer a grammar with the minimum number of syntactic productions, regardless of its class and nonterminal vocabulary.
3. Provide a first-order formula that specifies all and only the strings of the language.

Solution to part 1.

The language L is recognized by the deterministic finite automaton shown in Figure 1-123.

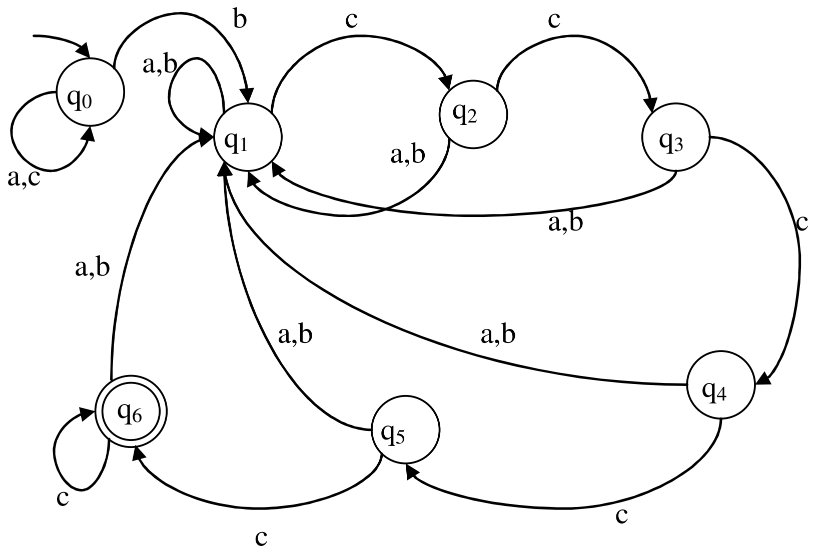


Figure 1-123 FSM recognizing the language L defined consisting of all and only the strings belonging to {a, b, c} * containing at least one be ending in 5 consecutive c.

Solution to point 2.

Since L is recognized by a finite automaton, there is certainly a regular grammar that generates L. However, the following grammar has a lower number of productions.

S → AbAccccc

A → aA | bA | cA | ε

Solution to point 3.

$x \in L \leftrightarrow x \in \{a,b,c\}^* \wedge (\exists y, z ((x = y.b.z.ccccc) \wedge (y, z \in \{a,b,c\}^*)))$

The well-known formula that formalizes the membership $x, y, z \in \{a,b,c\}^*$ is omitted.

Exercise 1.7.34

Formally define an automaton with two queues, informally described as follows:

The automaton acts as a language recognizer operating on strings stored on a tape input, it has a finite state control unit and two memories managed according to the FIFO policy (queues).

Every move of the automaton consists of the following steps:

- it reads the input character, or it does not read nor move the head (ε-move)
- it reads the character on the front of each queue
- it reads the status of the control unit

(Based on data collected in the previous readings):

- it changes the state of the control unit
- it could move the head on the right of the input tape (If it is not an ε-move)
- it removes the character on the fron of each queue
- it writes a string at the end of each of the two queues

Define then the configuration of such automaton, the relation formalizing a move and the acceptance of an input string.

Show how a single queue automaton, defined in the same way as the previous but with a single queue, could simulate the behavior of a two-queues automaton. Show how it is possible to build a single-queue automaton recognizing the same language recognized by a given two-queues automaton.
It is not required to formalize the behavior of a single-queue automaton, it is enough to describe briefly but precisely how it has to act in order to simulate a two-queues automaton.

Solution

Following a traditional scheme already used to formalize other abstract machines, the two-queues automaton is formalized as follows:

$<Q, I, \Gamma, \delta, q_0, Z_0, F>$, where the symbols have the usual meaning.

$\delta: (Q \times (I \cup \varepsilon) \times \Gamma \times \Gamma) \rightarrow (Q \times \Gamma^* \times \Gamma^*)$ is the transitino function (δ has to be subject to the same restriction adopted in the pushdown automata to have a deterministic two-queues automaton)

A configuration *c* is defined as:

$c = <q, x, \gamma_1, \gamma_2>, q \in Q, x \in I^*, \gamma_1, \gamma_2 \in \Gamma^*$

The transition relastiohsip is defined as follows:

$c_1 = <q_1, x_1, \gamma 1_1, \gamma 1_2> \;\vdash\; c_2 = <q_2, x_2, \gamma 2_1, \gamma 2_2>$ if and only if

$x_1 = x_2, \gamma 1_1 = A\eta 1_1, \gamma 2_1 = \eta 1_1 \lambda 2_1, \gamma 1_2 = B\eta 1_2, \gamma 2_2 = \eta 1_2 \lambda 2_2,$

and $\delta(q_1, \varepsilon, A, B) = <q_2, \lambda 2_1, \lambda 2_2>$;

or

$x_1 = i.x_2$, $\gamma 1_1 = A\eta 1_1$, $\gamma 2_1 = \eta 1_1\lambda 2_1$, $\gamma 1_2 = B\eta 1_2$, $\gamma 2_2 = \eta 1_2\lambda 2_2$,

e $\delta(q_1, i, A, B) = <q_2, \lambda 2_1, \lambda 2_2>$

Finally, the string x is accepted by the automaton if and only if $c_0 = <q_0, x, Z_0, Z_0> \vdash^* c = <q, \varepsilon, \gamma_1, \gamma_2>$ with $q \in F$.

A single-queue automaton can simulate a two-queues automaton in the following way:

The queue of the singe-queue automaton contains a string $\gamma_1\$\gamma_2$, where \$ is a special symbol used to divide γ_1 e γ_2 each one representing the content of a queue of the two-queues automaton.

In order to simulate a move of the two-queues automaton, the single-queue automaton has to perform a macro-move consinsting of the following steps:

- read and memorize in a finite state memory the first charachter of γ_1;
- copy the rest of γ_1 at the end of the queue, divided from the previous part with the \$ symbol;
- read and memorize in a finite state memory the first charachter of γ_2 (right after the \$);
  At this point the single-queue automaton has all the information needed to simulate the move of the two-queue automaton.
- copy the rest of γ_2 at the end of the queue, divided from the previous part with the \$ symbol;
- copy the rest of γ_1 at the end of the queue, divided from the previous part with the \$ symbol and followed by λ_1,defined by δ;
- copy the rest of γ_2 at the end of the queue, divided from the previous part with the \$ symbol and followed by λ_2,defined by δ;
- remove the \$ symbol remaining at the front of the queue;
- shift of the reading head;
- State transition q1 $\rightarrow$ q2.

Exercise 1.7.35

Consider a device A emitting an instantaneous light signal when a button has been pressed for exactly k seconds. The pressure of the button for less than k seconds is ignored while if the button is pressed for more than k seconds the time-count starts again.

1) Assuming discrete time and k=4, build a grammar generating the language L on alphabet {p, n, s} defined as follows. L contains all the sequence of characters p, n and s such that *p* states the pressure of the button, *n* the lack of pressure on the button and *s* the emission of the signal. The character *s* has to appear only after a sequence of *k p*s in a row.
   The grammar has to be of the minimum power and has to be simple (With few non-terminal symbols and productions). If it not possible to have both of them in a single grammar provide two grammars, each one meeting one of the requirements.
2) Specify with a first-order-formula a device A removing the time-discreteness constraint and considering a generic value of *k*.

Solution to point 1.

A simple context-free grammar is the following:

S → AS | BS | A | B

A → pppps

B → n | pn | ppn | pppn

A regular grammar is the following:

S → pA | nS | p | n

A → pB | nS

B → pC | nS

C → pD | nS

D → sS | s

Solution to point 2.

Let the predicates PB(t) and S(t) describe respectively the pressure of the button and the emission of the signal at time *t*.

The specification is formalized by the following formula:

$$\forall t (S(t) \leftrightarrow (\forall t_1 ((t-k \le t_1 \le t) \rightarrow PB(t_1)) \wedge \forall t_2 ((t-k < t_2 < t) \rightarrow \neg S(t_2)))$$

2. Computability

Exercise 2.1.1

Demonstrate that the following function:

g(x) = 1 if I_{f_x} is finite

0 otherwise

is not computable.

Solution

The theorem is a simple consequence of the Rice theorem. Consider the set

I = {computable functions with finite image}.

Obviously, I≠∅ and I≠{computable function }, so the set S={x|$f_x \in$ I} isn't recursive, because of the rice theorem, f.i. its characteristic function, isn't computable.

Exercise 2.1.2

Demonstrate that, given a value z_o, the function:

g(x) = 1 if $z_o \in I_{fx}$

0 otherwise

isn't computable.

Solution

The theorem immediately comes from the Rice theorem, considering, for a given z_o, the set Z_o={x | $z_o \in I_{fx}$}, that is the set of the computable functions which have z_o in their image. This set isn't empty and it does not coincide with the computable functions set.

Exercise 2.1.3

Demonstrate that the function:

g(x, y) = if f_x coincides with f_y then 1 else 0

is not computable.

Solution

The demonstration is reached reducing a function to g, g_u, that we can prove is not computable. In other words, if *g* was computable then g_u would have been computable too, but demonstrating that g_u is not computable, it is proved that *g* is not computable too.

Consider the function u(x) which is undefined for every x value, that is $\forall$x u(x) = $\perp$. The function *u* is obviously computable: every Turing machine, that enter an infinite cycle for every input value, computes the function *u*. Consider the index x_u of a TM that computes *u*.

Define the function g_u(x) that determine if a given TM, identified by its index, computes the function *u*. g_u is defined as the following

$g_u(x)$ = if f_x coincides with the function *u* undefined for every value then 1 else 0

Clearly the function g_u is a "specialization" of *g*: in fact $g_u(x) = g(x, x_u)$. So g_u is computable if *g* is computable too.

But it is easy to demonstrate that g_u is not computable , using the Rice theorem. Consider the computable functions set U by the only function *u* undefined for every x value.

$U = \{u \mid \forall x, u(x) = \perp\}$

Evidently U is not empty and it does not coincide with the computable functions set, so consider the set

$S = \{x \mid f_x \in U\}$

It isn't recursive, because of the Rice theorem. But g_u is the characteristic function of S, so g_u is not computable, and as a consequence *g* is not computable too.

Exercise 2.1.4

Show by diagonalization that the following functions are not computable:

1. $g(x, y, z)$ = **if** $f_x(y) = z$ **then** 1 **else** 0
2. $k(x)$ = **if** f_x è total **then** 1 **else** 0

Solution to point 1.

Let Us suppose that g is computable. Then $h(x) = g(x,x,0)$ is also computable: let x_0 be its Gödel number. Then $f_{x_0}(x_0) = h(x_0)$ = **if** $f_{x_0}(x_0) = 0$ **then** 1 **else** 0, that is obviously an absurd. Since the only hypothesis made is "g is computable", this hypothesis has to be false, that means that g is not computable.

Solution to point 2.

Obviously k is total. By absurd let us also suppose it to be computable. Define $g(x)$ = **if** $k(x)=1$ **then** $f_x(x)+1$ **else** 0. The function g is total, because k is total, and if $k(x)=1$ then f_x is total and $f_x(x)$ is defined, otherwise if $k(x) \neq 1$ then g is 0.

If k is computable, then g is computable too. Let n be the Gödel number of a TM that computes g: g is f_n, with $k(n) = 1$ (since g is total).

Then $f_n(n) = g(n)$ = **if** $k(n) = 1$ **then** $f_n(n)+1$ **else** $0 = f_n(n)+1$, that is absurd because $f_n(n)$ is defined, since f_n is total.

Exercise 2.1.5

Show that the function g defined as follows:

$g(x, y)$ = if $f_y(x) = 1$ then 1 else 0

is not computable.

Solution

The demonstration can be done directly, by diagonalization. Starting from *g*, let us define the function of an argument h, in this way:

$h(x) = \text{if } g(x, x) = 1 \text{ then } 0 \text{ else } 1$

If the function g is computable, then *h* is also computable, that is $h = f_{x_h}$ for some x_h. Let Us now compute the value $h(x_h)$ of *h* applied to the index of the Turing Machine that computes it.

$h(x_h) = \text{if } g(x_h, x_h) = 1 \text{ then } 0 \text{ else } 1$, (using the definition of g):

$h(x_h) = \text{if } f_{x_h}(x_h) = 1 \text{ then } 0 \text{ else } 1$, (since f_{x_h} is h):

$h(x_h) = \text{if } h(x_h) = 1 \text{ then } 0 \text{ else } 1 \Rightarrow$ contradiction!

Since we reach a contradiction, the hypothesis of computability of the function g is wrong.

Exercise 2.1.6

Prove directly, by diagonalization, that the problem of deciding whether $x \in L(M)$, for a Turing Machine M (over an alphabet I) and a string $x \in I^*$, is not decidable.

Solution

Suppose that the problem is decidable. Under this hypothesis, using an enumeration of the strings that belong to I* (where x_i is the i-th string of the enumeration), the language

$L_1 = \{ x_i \mid x_i \notin L(M_i) \}$

is decidable, and then there exists some Turing Machine M_j, that accept it. (the machine M_j works as follows: receiving x_i it produces M_i, check whether $x_i \in L(M_i)$, and accept x_i if and only if $x_i \notin L(M_i)$.)

Let Us apply the machine M_j to the string x_j:

- If M_j indicates that $x_j \in L(M_j)$, since M_j accept the language L1, $x_j \in L_1$. This is a contradiction, in fact if $x_j \in L(M_j)$ then $x_j \notin L_1$, by the definition of L_1.
- Otherwise if M_j indicates that $x_j \notin L(M_j)$, since M_j accept the language L1, $x_j \notin L_1$. This is a contradiction, in fact if $x_j \notin L(M_j)$ then $x_j \in L_1$, by the definition of L_1.

In either case we reach the usual contradiction, that let us conclude that the initial decidability hypothesis is not correct.

Exercise 2.1.7

Show by reduction that the following functions are not computable:

1. $g(x, y) =$ **if** $x \in If_y$ **then** 1 **else** 0
2. $g(x) =$ **if** f_x is constant **then** 1 **else** 0

Solution of part 1.

By absurd, let g be computable. Let Us prove that, under this hypothesis, it would be computable a function whose non computability has already be proven. This happen reducing to g the function that establish the termination of the TMs: **if** $f_x(x) \neq \bot$ **then** 1 **else** 0.

Let h(x) = **if** $f_x(x) \neq \perp$ **then** x **else** $\perp$. This function is trivially not computable: simulate the x-th TM with input x and if the TM ends then write x, else it does not end. Let n be the Gödel number of theTM that computes h ($h = f_n$).

Let Us define the function k(x) = g(x, n) = **if** $x \in If_n$ **then** 1 **else** 0 = **if** $x \in Ih$ **then** 1 **else** 0, that is computable, being g and h computable. By the definition of h(x) it is clear that $x \in Ih$ if and only if $f_x(x) \neq \perp$, that means that k(x) is equal to: **if** $f_x(x) \neq \perp$ **then** 1 **else** 0.

As a consequence of the hypothesis of being g computable, k would also result computable, but we know that's not true. Then the hypothesis must be false.

Solution of part 2.

Given a Gödel number x, let $g^x(z)$ = **if** $f_x(z) \neq \perp$ **then** 1 **else** $\perp$. Let p(x) be the function that associates each x to the Gödel number of g^x.

The function p is computable: consider a TM M that, given x, will build a TM for g^x, for instance via the UTM, and will compute its Gödel number. The function p is total, because the procedure works for every x. So $f_{p(x)}$ is a computable function (not necessarily total), with $f_{p(x)}(z)$ = **if** $f_x(z) \neq \perp$ **then** 1 **else** $\perp$. Let k(x) = **if** $f_{p(x)}$ is constant **then** 1 **else** 0, that is computable is g is so. However $f_{p(x)}$ is constant if and only if for each z $f_x(z) \neq \perp$, that means, if and only if f_x is total. Then k(x) is **if** f_x is total **then** 1 **else** 0, that we know being not computable by Exercise 2.1.4.

Exercise 2.1.8

A total ordering of the words of a language L over an alphabet O is said to be a *canonical order* if the words are in length-increasing order and the strings with the same length follow the typical lexical order (that derives by an appropriate total order over the alphabet O). For example, if a<b<c, a canonical order for the strings aaa, aba, cca, ab, bc, aaba is: ab, bc, aaa, aba, cca, aaba.

Now consider the definition of a language generated by a TM, given in the Exercise 1.3.7. Prove that:

1. L is generated by a TM if ,and only if,L is r.e.
2. L is generated by a TM in canonical order if, and only if, L is recursive.

Solution of part 1.

At first demonstrate that if L is *generated* by a TM M_1 then L is r.e., and thus is it is *accepted* by a TM M_2. M_2, that has an input tape and two extra memory tapes respect to M_1, copies the input on one of the extra tapes A and then simulates M_1 using the other extra tape B as the output. Every time that a string is generated on B, M_2 compares it with the one contained in the input (stored in A), and ends accepting if the two strings are the same. Otherwise, it continues with the following word. It is clear that if the string x is in L, sooner or later it will be generated on the tape B, and so M_2 will accept x. If the string isn't in L,then M_2 does not end, and so M_2 does not accept x.

Let Us now show that if L is a r.e. (that means that it exists an M_2 that accepts L), then it exists a TM M_1 that produces L.

A **wrong** proof of this is the following: M1 produces all the words of O^* and for every produced word it simulates the TM M2 that accepts L: if $x \in L$, M2 prints x, otherwise it goes on. The error in this demonstration is due to the fact that M2 could not halt for a given string x of O^* (generally it is well known that it is not possible to transform any TM into a TM that halts for every input): in this case M2 does not produce anything since that given time, even if there still are some strings of O^* that haven't been analyzed yet and that are in L.

The **correct** demonstration uses a canonical order over O^* and exploits a *pairs generator* to run the simulation of M1 for a given number of steps, in order to avoid that M2 stops producing, after having produced a string on which M1 does not stops. A pairs generator is a TM that produces a language made by *all* positive integer pairs, for instance (following the "well known" ordering often used to demonstrate the enumerability of the set of rational numbers):(1,1), (1,2), (2,1), (3,1), (2,2), (1,3), (4,1), (3, 2), (2,3), (1,4) ...

TM M1 simulates the pair generator, getting at each step the pair (i, j). M1 simulates for j steps the TM M2 that accepts L; if, exactly at the j-th step, M1 accepts the i-th position string in the Σ^* canonical order, then M1 writes the string on the output tape. M1 keeps on forever producing pairs and checking the previous condition.

If the x string in k position in the canonical order is accepted by M1 in w steps, then when we reach the pair (k,w) (surely produced), the M1 simulation for k moves with x as input brings us to the acceptance of the string, thus x is written on the output tape.

Solution of part 2.

The proof shows that if L is generated by a TM M in canonical order then it is recursive. Let L be infinite. Let us build a TM that simulates M and compares the input x with the string produced each time: if the generated string is equal to x then accept, if the generated string comes after x in the canonical order then stop without accepting, if the generated string comes before x in the canonical order then continue. The termination is guaranteed, since sooner or later x or a string longer than x will be produced, because the language is unbounded. Then L is recursive.

If L is finite, the previous proof is not valid anymore, because M does not produce anything more, once L is completely generated: if the length of the input x is greater than longest string in L, then the TM does not halt. This case of a finite L is a trivial case, because a finite language is always recursive (there is a TM that accepts L and stops for all the inputs, but we do not know which TM). Thus it is not generally decidable if a language is finite or not, the previous demonstration just proofs that there *exists* a TM that accepts L, but it is not always known how to build it.

In order to proof that if L is recursive then it is generable in canonical order, it is enough to create a TM M1 that produces in canonical order every string x in O^* and simulates over x the TM M2 that accepts L. If M2 accepts, then M1 prints the generated string and M1 continues. Since L is recursive, M2 may be always built in order to stop for all the inputs, so M1 generates all the words in L in canonical order.

Exercise 2.1.9

Let FC be the set of all the functions that, where defined, has constant value. For example, the following function belongs to FC:

f(x) = 5 if x even

f(x) = ⊥ if x odd.

(FC includes the totally undefined function).

1. Explain why the set of the indices of the Touring Machines that compute FC functions is not a decidable set.
2. Check whether it is a semi-decidable set. Justify.

Solution to point 1.

It is not a decidable set, as a consequence of Rice's Theorem.

Solution to point 2.

Let the set of the indices of the Touring Machines that compute FC functions be S.
The complement of S is semi-decidable. In fact it is possible to enumerate all the function values in a diagonal way, gradually increasing the number of simulated moves: a machine can simulate x moves of the machine that computes f(y). If f is not a constant function then there exist at least two values y_1 and y_2 such that $f(y_1)$ and $f(y_2)$ are both defined and $f(y_1) \neq f(y_2)$; so increasing alternatively the values of y_1 and y_2, sooner or later y_1 and y_2 will be discovered. This means that the complement of S is semi-decidable. Then S cannot be either semi-decidable, because otherwise S will be decidable.

Exercise 2.1.10

Say, justifying your answer, whether the following problem is decidable or not:

Given a Turing Machine M_1 that computes the function f(x) defined over the set of natural numbers, and given a Turing Machine M_2, check whether M_2 computes the (integer) square root function of f(x) $\sqrt{f(x)}$

Solution

The problem is undecidable for the Rice Theorem. In fact the function $\sqrt{f(x)}$ is a set of computable function (of cardinality 1). Then the set of the indices of the Touring Machines that compute $\sqrt{f(x)}$ is not recursive, that means that it is not decidable if a generic M_2 belongs to this set.

Exercise 2.1.11

Consider a generic TM M, with a single tape which is left-finite and right-infinite.Check if the following problems are decidable:

a. $\varepsilon \in L(M)$.
b. M has 10 states.
c. M with alphabet {0,1,b,/}, stated with empty tape, writes 3 times in a row the symbol 1 on the tape.
d. M, started with empty tape, visits every cell of the tape at least once.

Solution to point a.

Assuming that the symbol 0 is the encoding of ε, that f_i is the function computed by the machine M_i and $\varepsilon \in L(M_i) \Leftrightarrow f_i(0) \neq \perp$. Then $\varepsilon \in L(M)$ is a decidable problem if and only if the function h(x) = **if** fx(0) ≠ ⊥ **then** 1 **else** 0 is computable (where x is the Gödel number of M). According to the Rice theorem the h function is not computable.

Let us explain in a more detailed way the reasons that make us deduce that h(x) is not computable. Consider F as the set of functions f_i such as $f_i(0) \neq \perp$, and consider S the set of Goedel numbers of functions ∈ F (Figure 2-1).

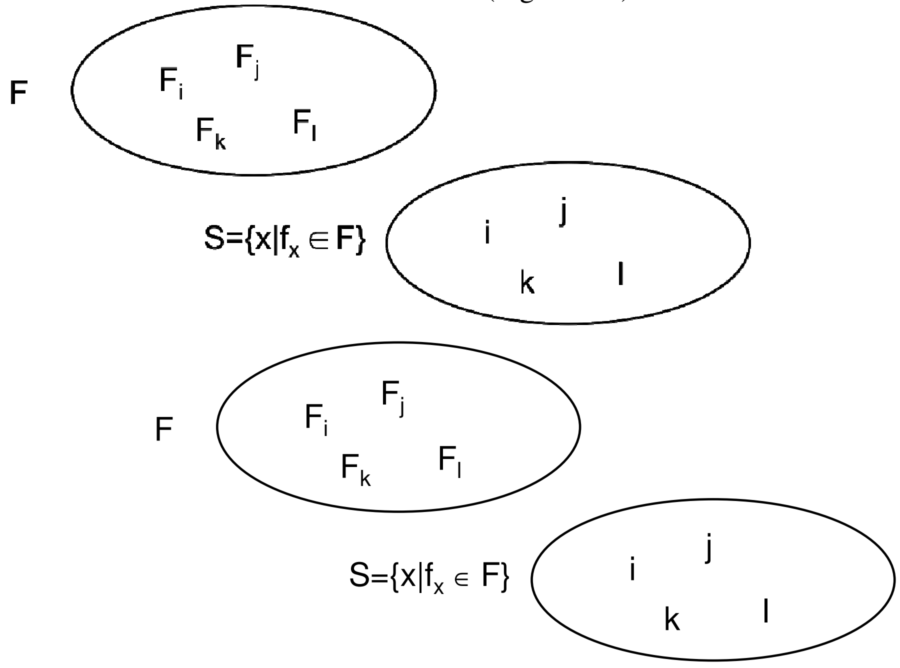


Figure 2-1 Set F of functions fi such that $f_i(0) \neq$□, and set of Goedel numbers of these functions.

According to Rice theorem S is recursive (that is the characteristic function of S is computable) if and only if F = ∅ or F = set of all computable functions.

It is easy to prove that F includes at least one function(e.g. the constant function of value 1) and it does not contain all of them (e.g. the totally undefined function). The consequence is that S is not recursive, that is ∀x if x ∈ S then 1 else 0 (characteristic function of S) is not computable.

So ∀x if fx ∈ F then 1 else 0 (characteristic function of F) is not computable.

By defition of F, ∀x if fx(0) ≠ ⊥ then 1 else 0 is not computable, that is ε ∈ L(M) is not decidable.

Solution to point b.

Obviously the number of states is a decidable characteristic for TM machines.

Solution to point c.

The proof of indecidability can be developed reducing the given problem to the termination problem. Consider a TM M' that simules M on a null input. Build another machine M" that works like M' but it never prints three 1 in a row on the tape, apart when it reaches a final state. This can be easily done if every alphabetical symbol of M' is translated into a pair of symbols: 0 becomes 01, 1 becomes 10. M" simules the working of M', such that every move of M' corrisponds to a macromove of M", which consists of at most four moves (e.g. to read the i cell and write 0, M" has to read the 2i-1 and 2i cells and switch to 0 the 2i-1 e to 1 the 2i cell). It is clear that at the end of the n-th macromove, M" is in the same configuration of M' at the nth move, only in the condition of translating all the cell pairs of M" into single cell of M'. It cannot happen to have three 1 in a row during the computation of M": if there are three 1 in a row in M', then in M" there is the string 101010; indeed it is possible to define macromoves of M" in such a way that writing three 1 in a row on the tape never happens: e.g., if M" has to change 10 into 01, first it puts a 0 in the first cell, then puts 1 in the second one. Let Us build M" also in a way that when M' (and so M with an empty input) accepts, M" writes three 1 on the tape and then it stops. Then M" writes three 1 in a row, if and only if, M stops with a null input, that is, if and only if $\varepsilon \in$ L(M). If the property were decidable, then this problem would be decidable too, but according to part (a) of this exercise, this is not possible.

Solution to point d.

The problem is decidable, under the condition that TM's tape is oneside-finite. Consider *i* the position of the given cell. A TM M', that receives as input the Gödel number of M, can simulate M, storing all the configurations reached everytime; M' stops and returns 1 (i.e. accepts) if, before reaching the i cell , M stops or falls into a configuration already reached. In the last case, in fact, M falls in a endless loop without leaving the portion (0,i-1).

M' stops and returns 0, instead, in the case M goes through the i cell .

It is clear that M sooner or later will reach one of the previous situations, because the number of possible configurations, in which the head is into the portion (0, i-1) of the tape,is finite.

Exercise 2.1.12

The following problem is known as the *Hilbert's Tenth problem*:

Given a polynomial P with integer coefficients in integer variables (for instance $5xy + 23z^3w^5 - 13x^8$). Check whether there exist integer roots for the polynomial, that means integer solution for the equation P (x, y, z, w) = 0.

After many years from its formulation, Hilbert's Tenth problem was shown to be undecidable. Check whether it is at least semi-decidable, justifying your answer.

Solution

Hilbert's Tenth problem is semi-decidable. In fact, let n be the number of variables of a given polynomial, it is possible to recursively enumerate all the n-pairs of integer numbers (for instance a recursive enumeration of integer pairs is the following one: {<0,0>, <1,0>, <0,1>, <0,-1>, <1,1>, <-1,0>, <2,0>, <-1, 1>, <1, -1>, <0,2>, <0, -2>, ...}).For every pair, it is easy to verify if it is a root for the polynomial. Thus, if there exists a radix, sooner or later it will be found by the former computation. However this computation will never end if do not exist any integer root of the polynomial.

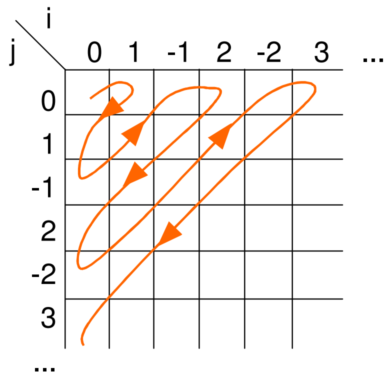


Figure 2-2 Diagonal enumeration of integer pairs.

Exercise 2.1.13

A formal grammar is said to be *context sensitive* if the length of the right part of all its productions is greater than or equal to its correspondent left part.

a. Check whether the problem of deciding if a given string belongs or not to the language generated by a context sensitive grammar is decidable or not.

b. (**) Check whether the problem of context sensitive language *emptiness* is decidable or not (given a context sensitive grammar G, is L(G) empty?). Justify your answer.

Solution to point a.

The problem is decidable due to the fact that the derivations of a context sensitive grammar can never diminish the length of the produced string. In other words, if, for a given context sensitive grammar G,

$$\alpha \Rightarrow^* \beta, \text{ then } |\alpha| \leq |\beta|.$$

In order to decide if $x \in L(G)$ it is enough to enumerate all the strings of length $\leq |x|$ derivable from the axiom G. This enumeration can be obtained by enumerating a finite number of derivation (starting from the axiom). In fact, given a string α of length n, there exist a finite number of derivable strings from α of the same length n. This happens because the length of a string can not diminish by applying a derivation step and the total numer of different strings of length n is k^n, where k is the cardinality of G's vocabulary. Then, after a finite number of derivation steps, either a string of length > n or a string already produced is derived.

Solution to point b.

First of all we are going to show that the context sensitive languages are (algorithmically) closed with respect to the intersection operation, which means that there exists a grammar G, also context sensitive, that produces $L(G1) \cap L(G2)$. An algorithm like this can be obtained via a simple modification of the general algorithm that has a similar construction for grammars of any kind.

An algorithm like this usually creates a grammar G that initially derives strings of the type $w = X_1\$X_2$ such that $w \in L(G)$ if and only if $x_1 \in L(G_1)$ and $x_2 \in L(G_2)$ (x_i and X_i are obtained the one from the other replacing lower case letters with capital letters and vice versa); then, through some suitable derivations X_1 is transformed into x_1 and $\$X_2$ into ε if and only if $X_1 = X_2$.

A grammar creation like this is not directly useful for our purpose, since it implies the cancellation of some characters, operation forbidden for a context sensitive grammar. It can be adapted in a way that G produces a string of type $<X_1, Y_1><X_2, Y_2> ...<X_n, Y_n>$ if and only if $X_1X_2 ...X_n$ belongs to $L(G_1)$ (or, better, the string obtained from it replacing lower case letter withcapital ones) and similarly for $Y_1Y_2 ..Y_n$. Thus the string x is obtained from $<X_1, Y_1><X_2, Y_2> ...<X_n, Y_n>$ simply via rules of type $<A,A> \rightarrow a$ (notice that an element of type $<A, B>$ is a single symbol of the non-terminal vocabulary of G). The details about the construction are not difficult, if we keep in mind that, in order to describe strings of different length (during intermediate translation steps), it is possible to introduce non-terminal symbols of type $<A, \Lambda>$ where $A \in V_1$ and Λ means the absence of an element of V_2.

Afterwards, observe that the context free grammars are a particular case of context sensitive grammar. Thus the intersection of two context free grammar is a context sensitive language. Thus, if the emptiness of context sensitive languages would be decidable, then it will be decidable also the emptiness of the intersection of two context free languages. However this last problem is clearly undecidable. Then the problem of the emptiness of context sensitive languages is undecidable too.

Exercise 2.1.14

1. Prove that, if $S=I_g$ where *g* is a computable ,total and strictly monotone function (that is $x < y \rightarrow g(x) < g(y)$), then S is recursive.

What does it change if:

2. *g* is computable, total, but only weakly monotone (that is $x < y \rightarrow g(x) \leq g(y)$)?
3. *g* is computable, total, only weakly monotone (thus $x < y \rightarrow g(x) \leq g(y)$), but asymptotically increasing(that is $\forall m \exists n (n > m \square g(n) > g(m))$)?
4. *g* is a monotone function but it is not total (that is $x < y \wedge g(x) \neq \perp \wedge g(y) \neq \perp \rightarrow g(x) < g(y)$)?
5. among the hypothesis of the previous point, the set $\{ i \mid g(i) = \perp \}$, where *g* is not defined,is decidable and $D_g = \{ i \mid g(i) \neq \perp \}$, the set where *g* is defined is unlimited (that is $\forall m \exists n (n > m \square g(n) \neq \perp \}$) ?

Solution of part 1.

Define an algorithm to decide S, that is to determine, given a generic *x*, if $x \in S$.

```
i := 0;
loop
   if g(i) = x then
        halt with success ($x \in I_g$)
   elsif  g(i) < x then
        i := i+1;
   elsif  g(i) > x then
        halt with failure($x \notin I_g$)
   fi
end loop
```

This algorithm is based on the fact that, being *g* monotone and total, if there exists an *i* such that $g(i) < x$ e $g(i+1) > x$, then $x \notin I_g$.

Solution of part 2.

If g is only weakly monotone the algorithm of the previous point does not work because if there is a moment in which g(i) ends its growth, remaining constant and < x, when i grows we cannot say if g(i) will continue to be the same (in this case $x \notin I_g$) or if it will start to grow again, becoming the same as x (so $x \in I_g$) or greater than x, without being the same (so $x \notin S$).

Solution of part 3.

But, if g is aymptotically increasing, then the previous algorithm can be used too, because it cannot happen that it remains constant, from a certain value of its argument on.

Solution of part 4.

If g is not total the algorithm obviously cannot be used, its steps, that permorm the computation of g are not executable (the computation of g would not end for the values of i such that $g(i) = \perp$).

Solution of part 5.

But if the set $\{i \mid g(i) = \perp\}$is decidable and $\{i \mid g(i) \neq \perp\}$ is unlimited then we can use the following variation of the algorithm, in order to decide

```
i := 0;
loop
if g(i) = ⊥ then
            i := i + 1;
elsif g(i) < x then
            i := i + 1;
elsif g(i) = x then
      halt with success(x ∈ Ig)
elsif g(i) > x then
            halt with failure (x ∉ Ig)
fi
end loop
```

In this algorithm the computation of g is avoided for values in which g is not defined. It uses the decidability of the definition set (the first condition of the if instruction is actually computable).

Exercise 2.1.15

A problem P_1 is said to be *more difficult* than a problem P_2 if it cannot happen that P_1 is solvable while P_2 is not. P_1 and P_2 are *equally difficult* if the solvability of P_1 implies the solvability of P_2 and vice versa.

a. Prove that the decision of containment between two sets ($S_1 \subseteq S_2$?) is more difficult than the equivalence decision ($S_1 = S_2$?).

b. Consider S a class of sets. Determine for which set operations the (recursive) closure of S, respect to these operations, ensures that the containment decision and the equivalence decision between two sets of S are two equally difficult problems.

Remember that a class of sets S is said to be recursively closed with respect to the operation *op* if : $\forall S_1,S_2$ ($S_1 \in S \wedge S_2 \in S \rightarrow S_1$ *op* $S_2 \in S$) and there exists an algorithm that can build S_1 *op* S_2 , given S_1 and S_2.

Solution of part a.

For every class of set S, if it is possible to decide the containment between 2 sets which belong to S, it is surely possible decide the equivalence too. In fact S1 = S2 if and only if $S_1 \subseteq S_2$ and $S_2 \subseteq S_1$. So we can use the same algorithm to decide both containment and equivalence. Note that, in general, it is not true the vice versa: there are set categories for which it is possible to decide the equivalence but not the containment.

Solution of part b.

If S is recursively closed respect to the union or intersection, the possibility to decide the equivalence between two sets of S ensures the possibility to decide the containment too. In fact $S_1 \subseteq S_2$ if and only if $S_1 = (S_1 \cap S_2)$ or if and only if $S_2 = (S_1 \cup S_2)$. So, if we want calculate if $S_1 \subseteq S_2$ for two generic sets S_1 and S_2, it is sufficient to compute ($S_1 \cap S_2$) (or $S_1 \cup S_2$) and decide if the result of this operation is S_1 (or S_2). Note that, to be precise, to validate the previous sentence, it has to happen that the class S is recursively closed respect to the considered operation, so it is not sufficient to know for every S_1 and S_2, $S_1 \cap S_2$ or $S_1 \cup S_2$ belong to S: it is necessary the existence of an algorithm to compute the result of these operations.

Exercise 2.1.16

Consider the language, LRAM, of the RAM machine without the conditional jump instructions.

Show if the termination problem for the LRAM$^-$ programs is decidable or not.

Solution

The problem is decidable. In fact, being the jump instructions without conditions (the only instruction is JUMP), there is only one execution flow for every program: so if a backward jump instruction is executed one time, the produced loop will not end. Otherwise it is mandatory to reach the end of the program because it would not be possible to execute every instruction more than once.

Exercise 2.1.17

Consider the following sentence A:

Given any possible positive integer x, if x is even divide it by 2. Otherwise multiply it by 3 and sum 1 to the result. Then divide it by 2. Repeat this procedure a finite number of times, soon or later we will obtain the value 1.

Show, if it is decidable the problem to determine if A is true or false, explaining the answer. (Note that: it is not asked to decide if A is true or false, but it is asked the decidability of its truth).

Solution

The problem is decidable. In fact, A is necessarily true or false. So its decision is equivalent to compute a constant function (the constant function 0 or 1), however computable. This does not mean that is possible to determine the truth of A: now day A is considered a property so likely true of natural numbers but it has not yet been proved (for what authors know).

Exercise 2.1.18

Prove that the theorems set L_{Tpa} of PA theory by Peano arithmetic is recursively numerable.

Solution

First prove a more general result: if a theory K allows a recursive set of axioms and has a language that is at most enumerable, then the set of theorems of K is recursively numerable.

It is sufficient to prove, as said in exercise 2.1.8, that theorems of K can be generated by a generative Tm. We can use the following procedure, called "The British Museum procedure" by Turing: we generate all possible formula sequences and for each of these we establish if it is a proof. If it true, we print the last formula of the sequence . Anyway, to obtain this result, we still have to solve an easy problem: if the language is numerable, then the alphabet is numerable too, while a TM alphabet is finite. We can conveniently code the alphabet symbols of the theory, e.g. representing the predicate A^n with n symbols A. In this way we can use a finite alphabet.

It is clear that the set of formulas written in the language of K is recursive, because of the simplicity of making rules of formulas themselves. So we can generate all the formulas of the language K, conveniently coded, with a TM. Prove now that the language D_K , made of all the sequences of formulas in the language K, is r.e. (it is even recursive): e.g. first we could generate all the sequences of length 1, made only by formulas of length 1, then all the sequences with max length 2, made by formulas of length 2, and so on. The fact that the TM's alphabet is finite ensures that at each step we can generate at most a finite number of formulas, and the method includes all the formulas: D_K is r.e.. Consider M the TM that generates D_K. The property of a sequence of formulas to be a proof of the last formula in the sequences is clearly decidable, because of the simplicity of the operations involved in deductions, on condition that it is decidable if the formula is an axiom, that in effect is one of the hypotheses. So we can realize a TM M' that checks if a sequence of formulas given in input is a proof and that ends for every input. Consider M" a generative TM that simulates on a memory tape a generation of a sequence by M and then simulates M' on each sequence little to little they are generated. If M' accepts, then M'' writes in output the last formula of the sequence (which is a theorem), otherwise it writes nothing. So M" generates all the theorems of K, because all the sequences of formulas are considered and so also all the proofs.

To end the exercise it is enough to consider that the Peano's arithmetic theory (but also the predicates computation) satisfies the condition of the previous results, and so the set of theorems of PA is r.e.

The hypothesis of decidability of axiomization of the previous result is important because that decidability allows checking mechanically if the proof is correct.

Observe that the set of valid formulas in standard model of Peano's arithmetic (i.e. in the common interpretation of arithmetic) is not r.e. According to the previous result, any decidable axiomization of arithmetic, like the PA one, lead to a r.e. set of theorems, so a different set (a proper subset, because all the theorems are valid) from that of valid formulas of PA. Then with any chosen axiomization for the arithmetic, if it is decidable there are true formulas of arithmetic for which it does not exist a first order formal proof, this sentence could be considered another version of the incompleteness theorem by Gödel.

Exercise 2.1.19

Exam of September 9 1998

Show, justifying the answer, which of the following sentences are true:

1. It is decidable the problem to determine if two C programs compute the same function;
2. It is semi-decidable the problem to determine if two C programs, for every input value we know a priori the termination, compute two different functions.
3. It is semi-decidable the problem to determine if the functions computed by two C programs, we know a priori that their definition domains are the same, are different.

Solution

Sentence 1 is false. In fact, 1 is a more general problem than determining if a problem (or, equivalently, a Turing Machine) compute a given function. This is problem is widely undecidable based on Rice theorem.

Sentence 2 is true; in fact it is sufficient to enumerate, with typical diagonal technique, the two programs' computations for every input data and for growing length: if there exists an input value for which results are different and moreover defined, it sooner or later will be found (in fact all the computations which ends sooner or later will be detected and it can not happen that an input value causes termination of one program and not of the other one).

Sentence 3 is obviously true because the problem is a particular case of the problem that refers to sentence 2.

Exercise 2.1.20

Demonstrate that the function:

(1) $l(x) = \text{if } f_x(x) \neq \perp \text{then } 1 \text{ else } 0$

is not computable.

Solution

The demostration is reached by diagonalization as follow. Assume by absurd that l(x) is computable and define m(x)

(2) $m(x) = \text{if } l(x) = 0 \text{ then } 1 \text{ else } \perp$

If l(x) is computable, then m(x) is computable too. Let:

(3) $m(x) = f_n(x)$.

Consider l(n): that assume vale 1 or 0

$l(n) = 0 \leftrightarrow$(per la (1)) $f_n(x) = \perp \leftrightarrow$(per la (3)) $m(x) = \perp \leftrightarrow$(per la (2)) $l(n) \neq 0$. Contradiction.

$l(n) = 1 \leftrightarrow$(per la (1)) $f_n(x) \neq \perp \leftrightarrow$(per la (3)) $m(x) \neq \perp \leftrightarrow$(per la (2)) $l(n) = 0$. Contradiction.

The consequence is that the hypothesis of computability on l(x) is not true: l(x) is not computable.

Exercise 2.1.21

Show, justify the answer, if is decidable the problem to establish is a generic program, written in any program language, computes the trigonometric function sin(x) con the better approximation of 10^{-3}.

Solution

The problem in not decidable. In fact, when any convention is made to represent by approximate form the real numbers, the functions' set who approximates sin(x) with a defect of 10^{-3} is not in the empty set neither in the universal set. Then, by Rice theorem, is not decidable to establish if a program computes the function belongs to or not this set.

Exercise 2.1.22

Consider the following program Modula-2.

```
Module UNKNOWPROGRAM (INPUT, OUTPUT)
     var X, Y, Z: INTEGER;
      begin
            READ (X); READ (Y); READ (Z);
            while X >= 1 and X <= Y*Z - (Z+2*Y) + Z div (Y - 34 * Z)
                  do
                    Z := X *Y*Z*2*(X - 8*X*Y*Z*Z*10);
                    Y := X *Y*7*2*(X - 8*X*Y*Z*Z*10);
                    X := X *Y*Z*2*(X*X - 8*X*Y*Z*Z*10);
            end /*endwhile*/;
            while Z <= 1 OR X <= Y*Y - (Z+2*Y) + Y div (Y - 34 * Z)
                  do
                    Z := X *Y*Z*2*(X - 8*X*Y*Z*Z*10);
                    Y := X *Y*7*2*(X - 8*X*Y*Z*Z*10);
                    X := X *Y*Z*2*(X*X - 8*X*Y*Z*Z*10);
            end /*endwhile*/;
            while Y >= 1000 or Y <= Y*Z*Y-(Z+2*Y)+Z div (Y-34*Z)
                  do
                    Z := X *Y*Z*2*(X - 8*X*Y*Z*Z*10);
                    Y := X *Y*7*2*(X - 8*X*Y*Z*Z*10);
                    X := X *Y*Z*2*(X*X - 8*X*Y*Z*Z*10);
            end /*endwhile*/;
            WRITE (X); WRITE (Y); WRITE (Z)
      end UNKNOWPROGRAM.
```

Show, justify the answer, if is decidable the problem to establish the its own termination corresponding to input data x = 106354; y = 341986; z = 765423.

Solution

The problem is surely decidable because, for every input data any program ends or not. The decidability hypothesis of the problem is equivalent to computability of a constant function. But, needlessly is known the constant's value.

Exercise 2.1.23

We define with the expression "undecidable problem" a "not decidable problem"; discuss, justifying the answers, which of the following sentences are true or false:

a) All the semi-decidable problems are undecidable.
b) There exist semi-decidable problems, which are undecidable.
c) Some undecidable problems are not semi-decidable.
d) Some decidable problems are not semi-decidable.

Solution

Remembering that decidable problems' class is strictly included into the class of semi-decidable problems and that there exist not semi-decidable problems, we immediately obtain the following answers:

a) False
b) True
c) True
d) False

Exercise 2.1.24

Discuss, justifying the answers, if the following problems are decidable:

1. Given two regular languages, decide if their intersection is empty or not.
2. Given any two recursive sets, determine if their intersection is empty or not.

Solution

1. The problem is decidable. In fact regular languages are algorithmically closed with respect to the intersection: there exists an algorithm that, given two state finite automatas, builds another automata that recognizes the intersection of the two languages recognized by the two first automatas. Because of the fact it is decidable to determine if a regular language is empty, it is obvious the decidability of original problem.
2. The problem is not decidable. In fact context-free languages are recursive. It is known that it is not decidable to determine if the intersection of two context-free languages is empty or not. So, it is clear that it is not decidable the same problem for recursive sets which properly include the context-free languages.

Exercise 2.1.25

Part 1.

Prove that the following problem P1 is undecidable.

P1: Given two computable functions f and g determine if the equation f(x) = g(x) has solution.

Hint: put P1 in relation with the following problem P2.

P2: Given a computable function f, determine if there exists a value x such that f(x) = k, being k a fixed value.

Moreover consider that the other problem to determine if a computable function has an empty image set (that is not defined for any x value) is obviously not undecidable.

The suggested way is not the only one to resolve the exercise and the student could try other ways.

Part 2.

Discuss, justifying the answer, if the above problem P1 is at least semi-decidable.

Part 3.

Resolve the second part with another simplifying hypothesis that f and g are total functions.

Solution of part 1.

P2 is clearly a particular case of P1. So the undecidability of P1 is derived from proving the undecidability of P2.

Define the function h(x) = if f(x) ≠ ⊥ then k else ⊥, that is obviously computable by an algorithm as the following:

1. read x;
2. compute f(x);
3. write k.

There exists a value x such that h(x) = k if and only if there exists a value x such that f(x) is defined, that is its image set is not empty. So if P2 was decidable also the problem to determine if a computable function has an empty image set would be decidable too. But because of the fact the last one is known to be undecidable, P2 is undecidable too, and as a consequence P1 results undecidable.P2 è palesemente un caso particolare di P1. Dimostrando quindi l'indecidibilità di P2 ne deriva a fortiori l'indecidbilità di P1.

Solution to part 2 and 3.

The problem P1 is semi-decidable.The justification is obvious if f and g are total functions: in fact it is enough to enumerate all the values of x and to compute f(x) and g(x) for each of them.

But if it is not known a priori that f and g are total it is necessary to modify the previous procedure as follows:

For each value of x and y ≥ 0, enumerated by the classical diagonal order indicated in the following figure, calculate at most y steps of the machine that computes f(x) and y steps of the machine that compute g(x).

If both of them ends within y steps and the computed values are the same, stop with success. Otherwise go on according to the enumeration of the figure with a new value of the pair <x, y>.

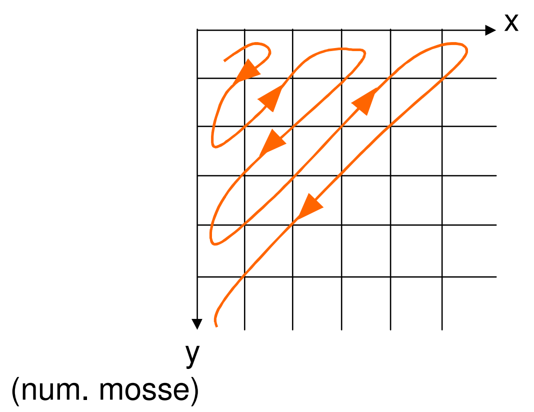


Figure 2-3 Diagonal enumeration.

The above procedure ends with success if and only if the equation f(x) = g(x) has a solution. Otherwise it goes on indefinitely.

Exercise 2.1.26

Prove if a program coded in Pascal, Modula2 or another similar programming language, is immune to the typical error of assigning to the index of an array a value not included between the established values**.**

Say justifying your answer, if this problem is :

a) Decidable;

b) Semi-decidable;

c) Not even Semi-decidable.

Solution

The problem is not decidable and neither semi-decidable.

In fact its complement is semi-decidable that consist in finding the presence of that error (that is to decide if exist a combination of input data which induce the program to commit at least one wrong indexing) . This problem is clearly semi-decidable, because it is possible to run the program with any input data: if the error exists sooner or later it will be found. The fact that the program might -with some input data - go in a endless loop it will be managed in the same way of the Exercise 2.1.25.. Since the complement problem is not decidable (as occurs with any other typical run-time error),the original problem cannot be even semi-decidable. In fact if they were semi-decidable, the problem and its complement would be both decidable.

Exercise 2.1.27

Prove, justifying your answer, if this problem is decidable:

There is a program P (coded in one of programming languages like C, Pascal,Ada,Modula2,...) and two variables x,y (integers) declared in P. Is it true that during whole execution of P x is every time >y (considering the fact that they were both initialized)?

Solution

The problem is undecidable. In fact we consider a generic program P'. We create another program P'' such that:
P'' includes all the variables of P' and moreover the variables x and y that are not included in P'. The executive part of P' is :

```
begin
  x := 2; y := 1;
  Parte esecutiva di P';
  x := 1; y := 2;
end
```

Then x>y during the whole execution of P'' if and only if P' does not end its execution. If it were possible to decide the problem, it would be possible to decide also the end of a generic program.

Exercise 2.1.28

Prove, justifying your answer, if is decidable the problem of establish if a generic Touring Machine M calculates a function f: $N \rightarrow N$ such that $\forall$ x, $x^2 + (f(x))^2 + 1 = 0$.

Solution

The problem is decidable. In fact no function, with the set of natural or integer numbers as domain and range, can have that property. So no one TM can calculate that unexisting function.

Exercise 2.1.29

Consider the following problem P:

Two trains proceed in opposite directions on parallel railways with constant speed V1 and V2 (km/h), respectively . They leave at time t_0 from two cities K km far away. Determine when they meet.

a) Say, justifying the answer, if the problem P is decidable.
b) Say, justifying the answer, if it is decidable determine if a generic program PC, coded in C, solves problem P.

Solution to point a.

Problem P obviously is decidable. It is the case of an easy problem of kinematics, solved by the formula:

t_i (time of the meeting, expressed in hours) = t_0 + K/(V1+V2)

Solution to point b.

The second problem is undecidable. Its undecidability is a typical consequence of Rice theorem; according this it is not decidable to determine if a generic algorithm computes a fixed function (that i sit solves a fixed problem).

Exercise 2.1.30

Say , justifying the answer, if the following problem is or not decidable:

Given two Regular Grammars G1 and G2 determine if the intersection of the generated languages is or not empty.

Solution

The problem is decidable.

In fact, we can algorithmically obtain from G1 and G2 two finite state automata A1 and A2 equivalent to them. So, always algorithmically, we can build a finite state automata A that recognizes the intersection of the two languages recognized by A1 and A2. at the end, as it is known, thank to pumping lemma it is possible to determine if L(A) is empty, verifying that a finite number of strings belong to it (those strings not longer than the number of its states).

Exercise 2.1.31

Say, justifying the answer, if the following problems are semi-decidable and, eventually, decidable:

1. Given the natural numbers x, y, z, determine if $y = f_z(x)$;
2. Same problem of the previous, but is known a priori that z is Gödel number of a Turing Machine that computes a total function.

Solution to point 1.

The problem is not decidable, as proved in Exercise 2.1.4. The problem is semi-decidable instead. In fact it is always possible to simulate the z-th TM through a universal TM: if $f_z(x)$ is defined then the universal TM can compute its value and answer to the given question.

Solution to point 2.

The problem is not only semi-decidable, but also decidable. In fact in this case, simulating the z-th TM we are sure that its computation ends and so we can always answer the question, in affirmative either negative way.

Exercise 2.1.32

Say, justifying the answer if the following problem is decidable:

Given two generic programs P_1 and P_2 which compute a polynomial of grade n fixed (e.g. n = 324), determine if P_1 and P_2 are equivalent.

Hint: take consideration about a straight line is located by two points, a parabola from three, etc.

Solution

A polynomial of grade n is located by n+1 points. To decide the equivalence between P_1 and P_2 is enough to give in input to both programs n+1 different values of the input variable: if (and only if) the corrispondent values produced by P_1 and P_2 are the same in every case then the two programs are equivalent.

Exercise 2.1.33

Say, justifying the answer if the following problem is decidable:

Given roadmap and two cities, determine if it exists a path that allows go from a city to the other.

Solution

The problem is decidable. In fact a roadmap can be represented through a graph (the vertexes correspond to the cities and edges to roads). The graph obtained obviously is finite and the problem to solve consists of determining if it exists a path that starts from a vertex to another. A lot of algorithms are known for this problem.

Exercise 2.1.34

Say, justifying the answer, if it is decidable (or semi decidable) the problem to determine if, given three programs P_1, P_2 and P_3 which compute function on integer numbers, P_3 computes the sum of P_1 and P_2 functions.

Note: formally the problem is to decide if given any three programs, which compute functions on integer numbers, the following sentence is true:

$\forall\ x_i,y_j\ \ P_1(x_1, \ldots, x_k)+P_2(y_1, \ldots, y_m) = P_3(x_1, \ldots, x_k, y_1, \ldots, y_m)$ $(1\leq i \leq k,\ 1\leq j \leq m)$, $x_i \in Z$, $y_j \in Z$.

Solution

The problem is not decidable, being a variant of the problem to determine if a given function (P_3) is equivalent to another one (P_1+P_2).

The problem is not semi decidable too, because its own complement semi decidable. To prove this last point we can proceed as in Exercise 2.1.25.

Exercise 2.1.35

1. Say, justifying the answer, if it is decidable the problem to determine if, given two generic programs written in C, which work on integer numbers, and it is known a priori the termination for every input data value, it exists a subset not empty of input values such that the two programs produce the same results for these values.
2. Say if the previous is semi decidable.

Solution to point 1.

Denote with P_1 and P_2 the two programs.

Assume that P_2 computes the constant function z_0. Evidently, if the problem results undecidable for this particular case, it will be undecidable in the general given case too.

The problem is to define a program that is able to compute a function

if $z_0 \in I_{P1}$ then 1 else 0

this problem does not exist (as proved in exercise 2.1.12), so the problem is undecidable.

Solution to point 2.

The problem is semi-decidable. P1 and P2 can be executed for every possible input data for which programs end. In this way sooner or later we will obtain an affirmative answer (that is the same results) if this exists.

Note that it is not necessary to use an diagonal enumeration technique (as that shown in Figure 2-3) to go through the context plain of two programs and their own possibilities (to execute y steps of each program in the x-th context) because we know a priori for which input data values the programs end.

If the contexts of the two programs are made of n-uplas of integer numbers it can be used the diagonal technique to enumerate contexts (as shown in Figure 2-2, in the case of pairs of integer numbers).

Exercise 2.1.36

Given an finite alphabet I, establish if the following problem is decidable: given a deterministic finite state automaton $A = (Q,I,\delta,q_0,F)$, and a string $x \in I^*$, verify if exists a string $y \in L(A)$ such as x is a substring of y.

Notice: a string $x \in I^*$ is a substring of a string y if y could be divided in such a way $y = v\ x\ w$, where $v, w \in I^*$. If the problem is decidable, establish the complexity, in term of number of states of automaton and of length $|x|$, of a decidable algorithm. You don't have to find a particular algorithm, neither details of its. To compute the complexity, it is enough to find a reasonable limited superior.

Solution

Find in A a set Init of achievable states from initial state: $q \in$ Init iff $q \in \delta^*(q_0,w)$ for some $w \in I^*$.

Also find a set Fin of states which we can achieve final state: $q \in$ Fin iff $\exists\ q_f \in F$ □ $q_f \in \delta^*(q, w)$ for some $w \in I^*$. This setting can be done in $\Theta(n^3)$, for example computing transitive closing of the graph of transition function. An alternative way to find the two sets is to minimize the automaton, already in time $\Theta(n^3)$: the resulting automaton from minimization is such that all states in Init and all states, except final states in Fin.

Found Init and Fin, can construct |Init| automaton $A_i = (Q, I, \delta, q_i, \text{Fin})$, for each $q_i \in$ Init, where $|\text{Init}| \leq n$. There are no more than n automatons and necessary time to construct each automaton $\Theta(n)$ (reduce to $\Theta(1)$ always reutilize the same automaton). We can now use the string x as input for each of these: If at least one automaton accepted, x is a substring of a string in L(A); If not accepted, x is not a substring of any string in L(A). Complexity is $\Theta(n\ |x|)$. Hence, the solution is $\Theta(n^3 + n\ |\mathbf{x}|)$.

Exercise 2.1.37

Say if the following problems decidable:
Given a function f, domain and co-domain N (same with natural number), calculated by the i-th Turing machine Mi that has the following properties:

$$\forall x \Big((\neg(x >= 0) \rightarrow f(x) \neq 5) \wedge$$

$$\Big((x >= 0) \rightarrow \Big(f(x) > 37 \vee \Big((\neg\, (f(x) = \perp) \rightarrow (f(x) < 100)\Big)\Big)\Big)\Big)$$

Solution
The formula must satisfy if f is always true. Indeed:

$(\neg(x >= 0) \rightarrow f(x) \neq 5)$
And true because **¬(x >= 0)** and false for every x in N.

$$((x >= 0) \rightarrow (f(x) > 37 \vee ((\neg (f(x) = \perp) \rightarrow (f(x) < 100)))))$$

Is also true because x is always >=0
If f(x) **f(x) = ⊥)** is true in second brand of ∨ (because the antecedent is false)

If f(x) ≠ ⊥) and f(x) >37 is true in first branch of ∨
If instead (f(x) ≠ ⊥) and f(x) <= 37, is true at second branch of ∨ because f(x) <= 37 implies f(x) < 100

So each of Turing machine computes a function that has the properties specified and the problem is decided, hence decidable.

Exercise 2.1.38

Say, justifying the answer, if the following sentence is true or false:

"If a problem P is decidable, it is semi-decidable too, but if it is semi-decidable and its complement is not decidable, then P is not decidable. Otherwisse, if P is not semi-decidable its own complement is not decidable."

Solution

The sentence is true, because all its components are true:

- If P is decidable, it is semi-decidable too.
- If P is semi-decidable and its own complement is not decidable, P can not be decidable too because the decidable set are closed respect to the complement.

For the same reason it is not possible that P was not semi-decidable and its own complement was decidable.

Exercise 2.1.39

Say, justifying the answer, if the following sentences are true or false:

- Given two generic functions f_1 and f_2 defined on natural numbers, the problem to decide if $\forall x\ f_1(x) \neq f_2(x)$ is decidable. (the problem's data are the functions f_1, f_2, not x!).
- Given two generic copmutable functions f_1 and f_2, the problem to decide if $\forall x\ f_1(x) \neq f_2(x)$ is decidable.
- Given two generic polynomials P_1 e P_2 defined on natural numbers, the problem to decide if $\forall x\ P_1(x) \neq P_2(x)$ decidable.
- Given a computable function f and a polynomia P defined on natural numbers, the problem to decide if $\forall x\ f(x) \neq P(x)$ is decidable.

Solution
Point 2: False. The proof can be done by reducion to the case of theorem 2.8 (pg. 202 italian edition). Then: suppose that the problem to decide, for every couple of computable functions f1, f2 is $f_1(x) \neq f_2(x)$ for every x, is decidable. then, also is decidable the problem TOT to compute, given a computable generic function f1, if f1 is total, that is different from ∞ for every x. It is sufficient take the function $f_2(x) = \perp$ for every x, that is surely computable. But TOT is undecidable for the theorem 2.8. Then the also the considered problem is not decidable.

Point 1: False, reduction from point 2. In fact, is 1 was true, 2 will be too, that it consider a subset os function 1. But 2 is false, the the 1 is false too.

Point 4: False, similar reduction from point 2. In fact, the constant function $f_2(x) = \perp$ can be considered as a case (degenerate) of a polynomial. Alternatively, we can denote that a polynomial is a function surely computable, therefore it is possible to reduce the case 4 and 2.

Point 3: True. The problem to verify if the polynomial calculated from P_1 e P_2 difference does not allow real roots. Being P a grade n polynomial (where n is the maximum grade between P_1 and P_2), it will have a finite number of roots n, then the problem is decidable.

Exercise 2.1.40

Say, justifying briefly your answer, if the following statements are true or false: "if a problem P is semi-decidable but not decidable, then P doesn't belong to P, the set of all problems solvable by a deterministic algorithm in polynomial time.

Solution
The statement is true. Indeed, if a deterministic algorithm solves a problem P in polynomial time, then P is certainly decidable. Moreover because the computation of algorithm always terminates.

Exercise 2.1.41

Say, briefly justifying your answer, if the following problem is decidable:
Set S1 is sequence of integer numbers {2, 28, 999, 34, 1278, 7649, 100000} and S2 is sequence of characters {a, a, c, y, e, s, t, u, d, a, b, x, o, p, w}
Given a generic program P, coded in any programming language (C, C++, Java, Modul-2…), determine if P, receiving input S1 then produce S2 as result.

Solution
The problem is not decidable. In fact, the property of P that we want to decide is certainly possessed by some computable function defined on all possible input file but

not all. According to Rice theorem then it is not possible to determine if a generic algorithm (however encoded) computes a function with this property.

Exercise 2.1.42

Consider the following problem:

Given a program P that computes the function f_P: Z $\rightarrow$ Z (Z is the set of the integer numbers), establish if f_P (x) > 0 for all x.

Say, justifying shortly the answer, whether the problem is decidable, semi-decidable but not decidable, or semidecidable.

Solution

The problem is not decidable. In fact the set of functions in which the specified property is applied, is neither empty nor the universe set. So we apply the theorem of Rice.

However the complement of the problem is semi-decidable. In fact, with the typical technique of diagonal enumeration, it is possible to find x, if it exists, such that f_P (x) $\leq$ 0.

Consequently the original problem cannot be semi-decidable, otherwise it would be also decidable.

Exercise 2.1.43

Consider the following grammar:

<Exp> $\rightarrow$ x | (<Exp> + <Exp>) | (<Exp> - <Exp>) | (<Exp> * <Exp>) | <Cost>*<Exp>

<Cost> $\rightarrow$ 0 | 1 | 2 | … 9

And consider the following problem:
given two expression E1 and E2 generated from the above grammar. Establish if $\forall$x(E1 = E2)
(x is interpreted as real variable and operation symbols as usual arithmetic operations.)

Say, briefly justifying your answer, if the above problem is decidable, semi-decidable but not decidable, or semi-decidable.

Solution
The grammar in question generates expression as:
c_1* x*(c_2*x – c_3*x)*(…)-…+(…),
where ci denotes coefficients number between 0 and 9. Computing (also algorithmically) such expressions,we obtain polynomials with integer coefficients in

variable x: for example $5x^5 + 18x^4 – 45x$ (NB: we do not obtain all possible polynomials in x: for example x + 5 and 17x are not obtainable from the expressions generated by grammar).

In general, two polynomials with integer coefficients are identical (they have the same value for each x) if and only if the coefficients of all monomials that constitute them match for every degree of monomials. Consequently, the problem is decidable.

Exercise 2.1.44

Say, justifying your anwser, if the following statements are true or false:

1. A necessary condition for a problem P to be undecidable is to be formalizable as a problem of computing a function f_P with an infinite domain.

2. A sufficient condition for a problem P to be undecidable is to be formalizable as a problem of computing a function f_P with an infinite domain.

Solution

The first statement is true: in fact if a problem was formalizable as computing a function f_P in finite domain, function f_P would be surely computable. (for example by means of a table that defines all of its values). Obviously this does not mean that an algorithm to compute f_P is available.

The second statement is false: in fact, for example, the function f(x) = x+2 defined over natural numbers has infinite domain but is obviously computable.

Exercise 2.1.45

Consider the two following programs:

```
#include <stdio.h>
main()
{
        int  a, b, sum;
        scanf("%d%d", &a, &b);
        sum = a + b;
        printf("Sum of a+b is:\n%d \nGoodbye\n", sum);
}
#include <stdio.h>
main()
{
        int  a, b, sum,;
        scanf("%d%d", &a, &b);
        sum = ( (a + b) / 3)*3;
        printf("Sum of a+b is:\n%d \nGoodbye\n", sum);
}
```

Say, justifying your response, if the problem of establishing if the two programs are equivalent, is decidable or not.

Would your response change if input variables “a”, “b”, “sum of both programs were declared as float instead of int? Why?

Solution

The two programs are not equivalent. In fact, the integer division and the subsequent multiplication by 3 produce a result different from (a+b) if this value is not a multiple of 3. Since the problem is solved,meaning decided, it is decidable too.

If these variables would be real instead of integer, the solution of the problem would be more complicated. In fact the division by 3 could produce approximations of the result that determine different output for some input values, so the two programs will not be equivalent. However, some sophisticated compilers could automatically simplify the division and subsequent muliplication by 3 making the two programs equivalent. Anyway the problem still remains decidable, because it refers to two fixed programs, so they might be equivalent or not.

Exercise 2.1.46

Consider the following problem:

Given a program P that computes the function f_P: N → N (N is the set of natural numbers) establish if $f_P(x) \neq 10$ for all x in its domain.
Say, justifying briefly your anwser, if the problem is:

- decidable
- Semidecidable but not decidable
- Not even semidicidable.

Solution

The problem is not decidable. In fact the set of functions for which we can apply the specified property is obviously neither empty nor the universal set. Therefore we can apply the Rice theorem.

However the complement of the problem is semidecidable. In fact, using the typical technique of diagonal enumeration it is possible to identify an x such that $f_p = 10$, if such x exists.

Consequently, the original problem is not semi decidable too, otherwise it would be also decidable.

Exercise 2.1.47

Say, justifying the answer, which of the following sentences are true and which are false:

"Let P be a problem formalized like a function f_P, being the set N of natural numbers its domain and codomain. f_P is not computable.

If we reduce the domain of f_P to a finite subset of N,then:

1. f_P can be surely computed
2. f_P cannot be computed
3. f_P could become computable but it could also remain incomputable ".

Solution

Sentence number 1 is TRUE (therefore the other ones are false). In fact, a function defined on a finite domain is always computable because we can describe it with a finite table.

There is obviously a Turing's Machine that "computes" a finite table of values: it is the encoding of an Array.

Exercise 2.1.48

1. Consider the following problem: given three generic programs P_1, P_2, P_3, that compute, respectively, the three functions f_1, f_2, f_3, defined on the domain of natural numbers N, is $f_1(x) = f_2(x) + f_3(x)$ $\forall x \in$ N ? Say, briefly justifying your answer, if the above problem is decidable or not. **Remark**: P_1, P_2, P_3 are data of the problem and they are not initially fixed.
2. What changes if P_2 and P_3 are initially fixed, meaning they are two known programs and they do not change from case to case, and, given P_1, we want to know if the function f_1 computed by P1 is constant and identically equal to a $f_2 + f_3$ (meaning $\exists$ k$\in$ N ($f_1(x) = k$, $\forall x \in$ N), and $f_1(x) = f_2(x) + f_3(x)$ $\forall x \in$ N)?.

Solution

1. The problem is not decidable, because of the Rice Theorem: one of its special case is one in which f_2 and f_3 are fixed (for example $f_2(x) = x$, $f_3(x) = 5.x$). In this case the problem reduces to state whether a general program P computes the function f (x) = 6.x; {f} is neither the empty set nor the universal set.
2. In this case the problem becomes decidable or not depending on the values of P2 and P3. Indeed, if the function $f_2 + f_3$ is not a constant function, the set of functions computed by the possible values of P_1 that are constant and identically equal to $f_2(x) + f_3(x)$ is the empty set, then the problem is decidable. Otherwise this set is not empty (it is the set of functions f_1 such that $f_1(x) = f_2(x) + f_3(x) = H$, $\forall x \in$ N for some value of H) ,therefore the problem remains not decidable.

Exercise 2.1.49

Let A(x) and B(x) be any two *first order* formulas that contains the free variable *x* (plus other non free variables). The sentence "A(x) is decidable" is an abbreviation for "is there an algorithm which states whether, given a value of x, A(x) is true or false."

Say, shortly justifying the answers, whether the following statements are true or false:

Given any A(x) and B(x):

1. If A(x) is decidable and B(x) is decidable, then $A(x) \wedge B(x)$ is decidable;
2. If A(x) is not decidable and B(x) is decidable, then $A(x) \vee B(x)$ is decidable;
3. If A(x) is not decidable and B(x) is not decidable, then $A(x) \wedge B(x)$ is not decidable;
4. If A(x) is not decidable and B(x) is decidable, then $A(x) \vee B(x)$ is not decidable;
5. If A(x) is not decidable, then $\forall x\, A(x)$ is not decidable;
6. If A(x) is not decidable, then $\forall x\, A(x)$ is decidable;
7. If A(x) is not decidable and B(x) is not decidable, then $\forall x\, (A(x) \wedge B(x))$ is not decidable;
8. If A(x) is not decidable and B(x) is not decidable, then $\forall x\, (A(x) \wedge \forall y\, B(y))$ is not decidable;

Solution

3. True;
4. False: if B were false, decidability of $A(x) \vee B(x)$ would be only based on the decidability of A(x);
5. False: if B is $\neg$ A, $A \wedge B \equiv$ False, so it is decidable;
6. False: if B were always true, then $A(x) \vee B(x)$ would be true, too.
7. False: a close formula is always either true or false, so it is decidable, even if we don't know its truth value.
8. True, for what we say on 5.
9. and 8. are false for what we say on 5.

Exercise 2.1.50

Please state, briefly justifying the answers, if the following statements are true or false:

3. The cardinality of the set of recursively enumerable sets is greater than the cardinality of the set of recursive sets.
4. The cardinality of the set of not recursively enumerable sets is greater than the cardinality of the set of recursive sets.
5. The cardinality of the set of not recursively enumerable sets is greater than the cardinality of the set of recursively enumerable sets.

Solution

1. False: both sets are enumerable (even if the set of recursive sets is not *recursively* enumerable).
2. True: the set of not recursively enumerable sets is the complement, wrt the universe of all sets, of the set of recursively enumerable sets. The universe of all sets has not enumerable cardinality while its complement has enumerable cardinality, so the set of not recursively enumerable sets cannot be enumerable; conversely, the set of recursive sets is enumerable.
3. True, for the same reasons as point 2.

Exercise 2.1.51

Let $P \subseteq N$ the set of even numbers. For a generic $x \in N$ let Dx be the definition domain of the function fx, that is the set of values z such that $fx(z) \neq \perp$.Please answer to the following questions, briefly justifying your answer.

1. Establish if, for a generic x, Dx = P is a decidable problem.
2. Let DP be the set such that $\{x|Dx = P\}$. Is DP recursively enumerable?
3. Say, briefly justifying your answer, if the following statement is true: let S be a set of MT indexes such that $x \in S$ implies that fx is a constant function. Moreover S is such that for all constant functions k there exists in S a machine index that computes k. Then S is not recursively enumerable.

Solution

1. No. Indeed, given a generic x it is always possible to compute y such that

   fy(z) = **if** z = 2.w for some w **then** fx(w) **else** $\perp$. .

   Obviously Dy = P if and only if Dx = N. Hence, if the problem was decidable,it would be also possible to decide whether a function is total; but this problem is also undecidable.
2. No. Indeed let S be a set such that for all $y \in$ DP there exists at least one x in S such that, for all z, fx(z) = fy(2.z).
   We know that for all function h, total and computable, there exists $y \in$ DP such that for all z is fy(2.z) = h(z) e fy(2.z + 1) = $\perp$. Hence h = fx for some $x \in$ S. So S satisfies the hypotheses of the theorem 2.10 (see the textbook), therefore S is not recursively enumerable (RE).
   Note that if DP was RE then so would S: indeed, given an algorithm that enumerates the values y of DP, it could easily be modified so that each y of DP is transformed into an x such that fx(z) = fy(2.z). (observe that is sufficient to find *one* such x, and not *all* x)
3. The statement is false. In fact, you can recursively enumerate some MT that computes all and only the constant functions: just build a machine that computes the function identically equal to zero and calculate its index, then build a machine that calculates the function identically equal to 1 and calculate its index, and so on: this will generate indexes of machines that compute all and only the constant functions but not all of these indexes.

Exercise 2.1.52

Given $\mathcal{L}$, a generic class of languages. Consider the containment problem between two languages of $\mathcal{L}$: L1 $\subseteq$ L2 | L1, L2 $\in$ L ? The containment problem is undecidable in the following cases:

- $\mathcal{L}$ = context-free language (i.e. generated by some context-free grammar).
- $\mathcal{L}$ = recursively enumerable languages (i.e. accepted by some Touring machine).

Please state, justifying the answer, if it is decidable or not in the other following cases:

1. $\mathcal{L}$ = context-sensitive languages (i.e. generated by some contex-sensitive grammar). A *context-sensitive grammar* is a grammar in which any production are of the type $\alpha \to \beta$, where $|\alpha| \leq |\beta|$.
2. Regular languages (i.e. accepted by finite state automata) (**Optional**)

Solution

1. The problem is undecidable for context-sensitive languages. It is clear that context-free grammars are a special case of context-sensitive. It is therefore not possible that a problem that is more general than an undecidable problem is decidable.
2. The problem is decidable. Indeed regular languages are (recursively) closed with respect to all set operations. Now, L1 $\subseteq$ L2, if and only if L1 $\cap$ (L2^) = $\emptyset$. (L2^ indicates the complement of L1). For the above mentioned closure property L1 $\cap$ (L2^) is regular. The regular language emptiness is easily decidable (e.g. exploiting the pumping lemma).

Exercise 2.1.53

Please state, briefly justifying the answers, if it is decidable the fact that a functional procedure written in Modula-2(or Pascal, or Ada,...) has side-effects or not. Remember that a functional procedure has side-effects when the effect of its execution modifies more variables of the caller program than the one explicitly written on the left of the assignment symbol.

After this, briefly justifying the answers, please state whether the problem is semi decidable or not.

Solution

The problem is undecidable. Indeed using a technique suitable for many typical run-time properties of programs, consider a generic program called P. Then build a functional procedure F this way:

F contains all variables of P and the variable x (e.g. an integer) that does not appear in P but is in the environment of the program that uses (calls) F. F's structure is the following:

```
procedure F(): integer
    The declarative part of F contains as local variables all the variables of P;
    begin
        executive part of P;
        x:=1;
        return 10
    end
```

Consequently F has side effects (modifies the value of x) if and only if P's execution comes to an end. Then if it was possible to decide the main problem, it would be possible to decide about the termination of a generic program, too.

However, the problem is semi decidable. Indeed, it is possible to enumerate all of the "contexts" in which a functional procedure could be called (all the possible states of caller programs which can make a call to F). For all of these contexts it is then possible to simulate F's execution: if during its execution a side-effect is generated we can answer yes.

Adopting then a typical "diagonal" technique to cross all contests of F and the length of its possible executions (execute K moves of F's execution corresponding to the N-th contest) it is easy to obtain the positive answer if it exists.

Exercise 2.1.54

Please state, briefly justifying the answers, if the following statements are true or false:

1. If a grammar has only α ---> β productions such that $|\alpha| \leq |\beta|$ then the language it generates is recursive.

2. If a grammar has some α ---> β productions such that $|\alpha| > |\beta|$ then the language it generates is recursively enumerable.

3. If a grammar has some α ---> β productions such that $|\alpha| > |\beta|$ then the language it generates is recursively enumerable**,** but not recursive.

Solution

1. True. Indeed counting the productions according to increasing length one can also count all generated strings according to increasing length, being able to stop the production when a string longer than the one under exam has been produced.

2. True. All grammars generate recursively enumerable languages.

3. False. Indeed the following grammar generates the language L = {a}, obviously recursive.

S -----> AB

AB -----> a

Exercise 2.1.55

Let D be a *fixed* and *finite* subset of $\mathcal{N}$ (for instance D = {2, 55, 3, 688993, 9993, 12, 3452, 87654}).

Let F be a *fixed* subset of the set of the functions with domain and co-domain D, defined on whole D (for example, F = $\{f_1, f_2, f_3 | f_1(2) = 3, f_1(55) = 688993, \ldots f_2(2) = 55, \ldots. f_3(87654) = 3452\}$).

Let S be a *fixed set* of indexes of Turing machines that compute only total functions.

Say, briefly explaining your answer, if, it is decidable or not the following problem with fixed D, F, S:

Given a generic $x \in S$, $\exists f \in F$ such that $\forall z (z \in D \rightarrow f_x(z) = f(z))$?

Solution:

The problem is decidable. Given x, we can simulate the computation of f_x on all the values of D. Since f_x is a total function and D is finite this computation will end in a finite time. With this result, it is easy to prove that the graph of this function is the same, on D, as the one of a function that belongs to F. Remark that a total function on a limited domain is always computable because its graph is a limited set of points on integer Cartesian plane.

3. Complexity

Exercise 3.1.1

Describe a deterministic TM that accepts the language $L = \{a^n b^n \mid n \geq 1\}$, with a *spatial* complexity equal to $\Theta(\log n)$.

Solution

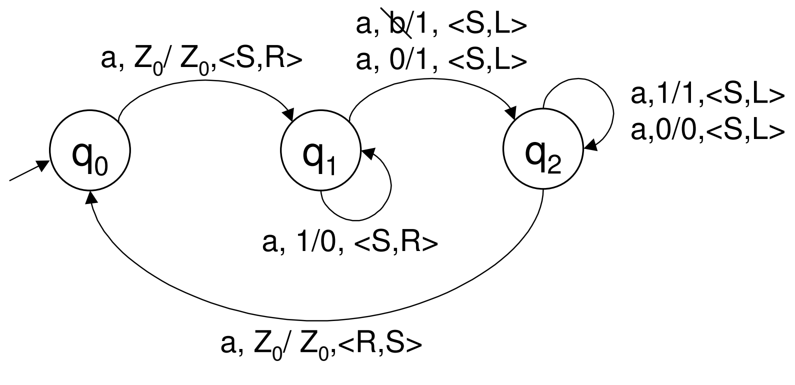
One of the TM for L saves a^n on its memory tape and then compares it with the string b^n given as input. The time complexity is 2n+1, while the spatial is *n* (if the length of the string is 2n). In order to reduce the spatial complexity, it is possible to use memory tapes as binary counters[2]

The method is applicable to a language L with a TM with 2 memory tapes as follows: count the occurrences of *a*, then the occurrences of *b* and, at the end, compares the counters (represented in binary). With a TM with one memory tapes, count the *a* and subtract 1 for each *b*. It is easy to verify that the asymptotic expression of time and spatial complexity remains unchanged. In the following, you will find the solution using a single memory tape TM, called M. We expect that the time complexity will decrease following a logarithmic factor.

The state set Q of M is {q0, q1, q2, q3, q4, q5, q6, q7, q8, qF} with qF as unique final state. The alphabet Γ of the tape of M is {0, 1, , Z0}. The counters are represented in binary from right to left; the LSB is the leftmost bit and the MSB is the rightmost. In the most significant positions there are characters b/ instead of 0.

The graph of the transition of a single tape TM is composed of three parts:

1. The first part, shown in **Figure 3-4**, performs a binary sum between 1 and the counter for each *a* given as input: from the state q_0 you can go to q_1 when there is an *a* as input; in q_1 there is the increment of 1 if the LSB is already 1, propagating the carry to right; from q_1 you move to q_2 switching to 1 the first bit of the tape, that it is 0 or blank (same meaning of a 0); in q_2, the machine comes back to the beginning of the memory tape and goes into q_0 ready to read a new input symbol, moving the head to right (during all the other operations the head was not working).
2. The second part, shown in **Figure 3-5**, subtracts 1 to the counter for each *b* in input: from the state q_0 the machine goes to q_3, when there is a *b* in input; in q_3 the machine subtracts 1 using the carry, if needed; from q_3 to q_4 the machine subtracts the carry, from the first bit equal to 1; in q_4 the machine verifies if there are other digits in most significant position: if this is true, the machine goes to q_6, otherwise the bit who was switched from 1 to 0 was the most significant and must be substituted with a blank (transition to q_5); in q_6, at the end, the head on the memory tape is moved to the beginning and the machine moves to q_7, that verifies if there are other *b* as input, moving M to q_3 if the answer is affirmative.
3. The third part, shown in **Figure 3-6**, verifies if the counter is equal to zero in the state q_7 and, if the answer is positive, accepts.



2 It is possible to prove that a TM model that exclusively uses memory tapes as counter (unary or binary) is equivalent to the general model of a TM, only if the memory tapes are, at least, two.

Figure 3-4 Part of the TM who stores, on the memory tape, the binary value of the number of *a*.

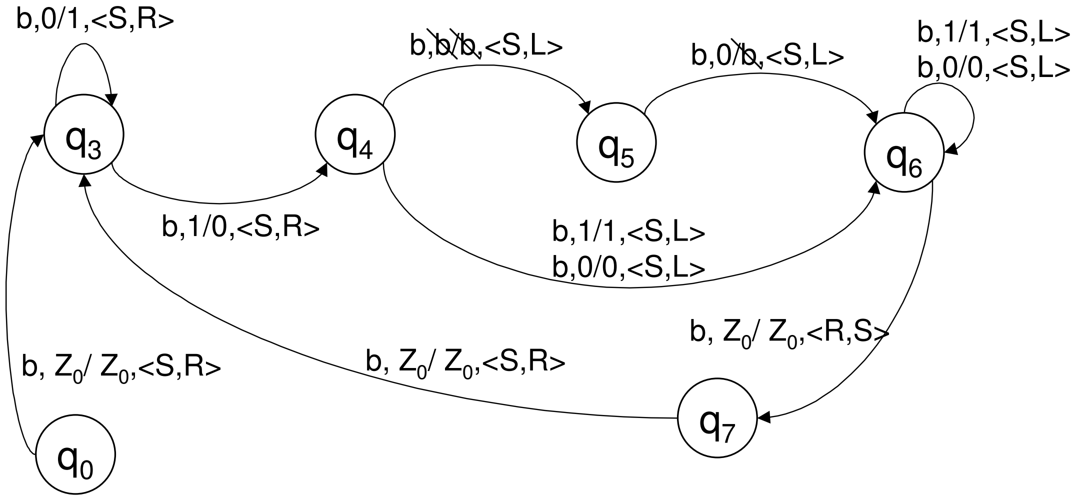


Figure 3-5 Part of the TM who subtracts 1 to the binary counter, saved on the memory tape, for each *b* as input.

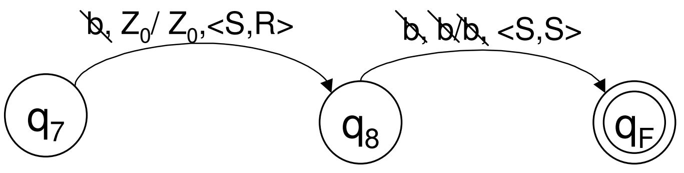


Figure 3-6 Part of the TM who verifies if the binary counter, saved on the memory tape, is null.

Now we evaluate the time and spatial complexity of M.

For the spatial complexity, it is easy to verify that, at most, the machine uses, if the input length is equal to 2, $\lfloor \log 2n \rfloor$ +2 memory cells, and the complexity is $\Theta(\log n)$.

For the time complexity is needed, after reading an even number of *a* equal to i: l(i) steps to sum 1 to the counter, l(i+1) steps to move the head to the beginning of the tape; after reading a number of *b* equal to i, it is needed: l(i) to subtract 1, l(i-1) to move the head at the beginning of the tape. In the worst case we have:

$$\mathrm{TM}(n) = \sum_{i=1}^{n}(1+l(i)+1+l(i+1)+1)+1+\sum_{i=1}^{n}(l(i)+2+l(i-1)+2)+1 =$$

$$= 2+8n+2\sum_{i=1}^{n}l(i)+\sum_{i=1}^{n}l(i-1)+\sum_{i=1}^{n}l(i+1) = 1+8n+2\sum_{i=1}^{n}l(i)+2\sum_{i=1}^{n-1}l(i)+l(n+1)$$

Since $\sum_{i=1}^{n}\lfloor \log_2(i)\rfloor = (n+1)\lfloor \log_2 n\rfloor - 2^{\lfloor \log n\rfloor+1}+2$, we obtain:

$\mathrm{TM}(n) = 1+8n+4(n+1)\lfloor \log_2 n\rfloor - 2^{\lfloor \log n\rfloor+3}+\lfloor \log_2(n+1)\rfloor+2$, that is $\Theta(n\log n)$.

Exercise 3.1.2 (*)

The relation Θ causes a partition in functions which is useful in the study of the complexity of the computation. Please define other relations, in which there are at least one thinner and at least one coarser than Θ, which you think are useful for the study of the complexity of the computation.
Explain the reasons of your choice.
Remember that R' is thinner than R if xR'y implies xRy for every x and y. R' is coarser than R if R is thinner than R'.

Solution

A relation Θ' thinner than Θ is the following:

f is Θ' of g if and only if $\lim_{n\to\infty}\frac{\phi(\nu)}{\gamma(\nu)}=1$.

This relation can be useful in the complexity analysis because includes only functions whose behavior to the infinite is very similar: for example f(n) = n+log(n) and g(n) = n are equivalent regarding Θ', but f(n) = 2n and g(n) = n it are not equivalent.
A relation Θ'' coarser than Θ is the following:
f is Θ'' of g if and only if there exist 3 constants, k, h1 and h2 such that for every $n > k$, $f(n) \le g(n)^{h1}$ and $g(n) \le f(n)^{h2}$.
This relation can be useful in order to distinguish, in first approximation, algorithms with polynomial complexity from algorithms with exponential complexity or algorithms with logarithmic complexity. For example, n is Θ'' equivalent to n^3 but $\Theta'' > \log(n)$ and $\Theta'' < 2^n$.

It is easy to demonstrate that both Θ' and Θ'' are equivalence relations and that they induce order relations on the respective equivalence classes.

Exercise 3.1.3

Write a RAM program that computes function f(n) = n! and evaluate its complexity with both *logarithmic cost criterion and constant cost criterion.*

Solution

We can build the following SMALG coding of n! computation:

```
begin
read n;
i := 1;
fatt := 1;
while i <= n
do
      fatt := fatt*i;
      i := i + 1
od;
write fatt
end
```

Without encoding the whole program in RAM language, it is clear that time complexity of such program is Θ(n) with the constant cost criterion and spatial complexity is Θ(1).

With logarithmic cost criterion the time complexity is Θ(n × execution cost of single cycle): the execution cost of single cycle is determined by *fatt := fatt*i*, so is proportional to the complexity. This instruction corresponds to the following RAM instructions:

```
LOAD 3      {3 corresponds to fatt}  {Cost : l(M[3]) = l(fatt)
                                      (the address of the instruction is constant)}
MULT 2      {2 correspond to i}      {Cost : l(M[3]) + l(M[2]) = l(fatt) + l(i)}
STORE 3     {fatt := fatt*i}         {Cost : l(fatt*i) = l(fatt) + l(i)}
```

Variables fatt and i assume, during the execution, values 1, 2, 6, ...n! and 1, 2, 3, ...n

Hence the total complexity of the cycle execution is $\sum_{\kappa=1}^{v} 3l(k!)+2\lambda(\kappa)$.

We can ignore l(k), dominated by l(k!),and we can prove that $\sum_{k=1}^{v} l(k!)$ is Θ(n^2 log(n)), in the same way as we can prove that log(n!) is Θ(n log(n)).

The spatial complexity is computed simply considering that the greatest number in memory is n!, so with logarithmic cost criterion we obtain Θ(l(n!)) = Θ(n log(n)).

Exercise 3.1.4

Evaluate the time complexity of the machine built in the Exercise 1.7.5.

Solution

The non deterministic pushdown automaton built in the solution of the Exercise 1.7.5 has a time complexity equal to Θ(n). In particular, it is absolutely equal to n: indeed, the automaton executes one steps for each input character.

Note that this complexity refers to a nondeterministic machine. We cannot say that the same complexity could be obtained also from a deterministic algorithm (for example a Turing Machine) which recognizes the same language.

In this case, it is easy to build a two tape Turing Machine which recognizes the same language with a time complexity equal to Θ(n): it is enough to copy the input on both tapes, move the head of the first tape to the beginning and perform a comparison, moving the first tape right and the second left.

Vice versa, the single tape Turing Machine who recognizes the same language (described in the Exercise 1.3.6) has a time complexity equal to Θ(n^2), because, for each pairs of deleted characters, it has to scan the entire tape. Therefore, it has to perform n steps for the first pair, n-2 for the second, n-4 for the third …

Exercise 3.1.5

Describe a RAM machine that reads an integer value $n \geq 1$ and computes the value n^n; evaluate the obtained time complexity. Create, besides, a solution without using MULT instruction, substituted by a cycle of addition, and evaluate its complexity.

Solution

It is possible to computes n^n with iterative multiplication. We use memory cell 1 to store n, cell 2 to store a counter, to be decreased at every multiplication, and cell 3 for every partial result. After loading *n* and initializing the register, we start a cycle of *n* multiplication, decreasing M[2] at every step, stopping when M[2] = 0. The cycle must guarantee at the end of every iteration:

$M[3] = M[1]^{M[1]-M[2]}$.

In this way, when M[2] =0 we have $M[3] = M[1]^{M[1]} = n^n$.

```
          READ 0           {read n from accumulator}
          STORE 1          {n}
          STORE 2          {counter := n}
          LOAD =1
          STORE 3          {partial result set to 1}
LOOP:     LOAD 1           {store n in the accumulator}
          MULT 3           {multiply partial result by n}
          STORE 3
          LOAD 2           {decrease counter}
          SUB =1
          STORE 2
          JGZ LOOP         {if counter >0, continue}
          WRITE 3
          HALT
```

Evaluation of the spatial complexity using uniform cost criterion: S(n) = 3.

Evaluation of the time complexity using uniform cost criterion: T(n) = 5 + 7n +2.

The uniform cost criterion does not fit this situation because n^n growths too fast compared to *n*: the most realistic criterion is the logarithmic one.

$T(n) = [l(n) + l(0)] + [l(n) +l(1)] + [l(n) + l(2)] + [l(1)] +[l(1) +l(3)] + \sum_{i=1}^{n} \big[[l(n) + l(1)] + [l(n) + l(3) + l(n^{i-1})] + [l(n^i) + l(3)] +[l(i) + l(2)] + [l(1) + l(i)] + [l(2) + l(i-1)] + [l(6) + l(i)]\big] + [l(n^n) + l(3)]$.

Considering only the main terms:

$T(n) \approx \sum_{i=1}^{n} (l(n^i) + l(n^{i-1}))$ that is $\Theta(n^2 \log n)$, because $\sum_{i=1}^{n} l(n^i) \approx \sum_{i=1}^{n} i\ l(n) = \frac{n(n+1)\log n}{2}$.

The spatial complexity could be easily computed considering that the biggest number stored in memory is n^n: S(n) is $\Theta(n \log n)$.

Without using MULT instruction, if SHIFT is not available, it is possible to use a cycle of addition in order to implement the multiplication.

A new counter is needed, it will be saved in M[4], and a new temporary variable wherein we can save the partial result, stored in M[5]. The MULT[3] operation is substituted by:

```
            STORE 4
            LOAD = 0
            STORE 5
MOLT        LOAD 5
            ADD 3
            STORE 5
            LOAD 4
            SUB = 1
            STORE 4
            JGZ MOLT
            LOAD 5
```

The spatial complexity does not change, while the time complexity has to consider the additional cycle.

Uniform cost criterion: T(n) is now 5 + n(4+7n+6) + 2, then $\Theta(n^2)$.

Logarithmic cost criterion: we compute the complexity considering the main operation ADD 3. At the *i-th* iteration of the external cycle and the *j-th* of the internal one, this operation costs l(M[0]) + l(3) + l(M[3]) = $l(j \cdot n^{i-1}) + l(3) + l(n^{i-1})$

Then $T(n) \approx \sum_{\iota=1}^{\nu}\left(\sum_{\varphi=1}^{\nu}(l(j \cdot \nu^{\iota-1}) + \lambda(\nu^{\iota-1}))\right)$.

Considering $\sum_{\varphi=1}^{\nu} l(j \cdot \nu^{\iota-1})$ $\Theta(n \log n^i)$, T(n) is $\Theta\left(\sum_{\iota=1}^{\nu} n \cdot \iota \cdot \lambda o\gamma(\nu)\right) = \Theta(n^3 \lg n)$.

If we would have the left-SHIFT operation (that corresponds, in the binary base, to the multiplication by two), the multiplication could be realized by a cycle of, at most, log(n) ADD and SHIFT operation, using the same method applied to hand-multiplication of two numbers. The complexity, in this case, becomes, considering the logarithmic cost criterion, $\Theta(n^2 \log^2 n)$. We can give an intuitive explanation of this.

The external cycle is executed n times and, for each iteration, it computes, at most, $\log_2(n)$ times SHIFT operation, followed by the ADD operation. Every ADD (as well the SHIFT) costs, at most, n*log(n). Total is n*(log(n))*n*log(n), that is: $\Theta(n^2 \log^2 n)$.

Exercise 3.1.6

Consider the following sorting algorithm. The input is composed of a sequence of *n* integer number, their values are between 0 and a *k* fixed value. The algorithm uses an array A of *k* counter, initialized to 0. While reading an integer value *j* from input, the counter in position *j* is increased by 1. The input sequence is scanned once; at the end of that, A[i] contains the number of occurrences of the element with value i in the input sequence. Subsequently, it writes onto the output tape, for i included in the interval from 0 to *k*, the value *i,* A[i] times. The result is an ordered permutation of the initial sequence.

1. Write the code of the algorithm in RAM machine language
2. Evaluate the time complexity in the worst case, using both the constant cost criterion and the logarithmic cost criterion, relative to *n*.
3. How could you implement the algorithm on a Touring Machine with any number of tapes, and which would be its time complexity? You are not asked to write actually the algorithm, but only write briefly how could be its implementation to minimize time complexity.
4. (*) Answer question c considering a Touring machine with only 3 tapes

Solution of point a

A RAM program that implements the algorithm described above is the following:

```
          READ     1 {N}        {It reads the number of data to read, named
N}
          LOAD=    0
          STORE    2 {CONT}     {CONT:= 0}
INIZ      LOAD     2 {CONT}     {while CONT≤ K it executes the cycle of}
                   {counters initialization }
          SUB=     K
          JGZ      LETT         {At the end of the cycle it starts to read}
          LOAD     2 {CONT}
          ADD=     10           {The array A of counters starts at the cell 10.
                                INDEX is always = CONT + 10}
          STORE    3 {INDEX}
          LOAD=    0
          STORE*   3 {INDEX}    {A[INDEX] := 0}
          LOAD     2 {CONT}
          ADD=     1
          STORE    2 {CONT}     {CONT := CONT + 1}
          JUMP     INIZ
LETT      LOAD=    1
          STORE    2 {CONT}     {CONT:= 1}
```

```
CICLOL   LOAD      2 {CONT}   {while CONT≤ N, it executes the read cycle
and }
                   {updates the counter }
         SUB       1 {N}
         JGZ       SCRITT     {At the end of the cycle, it starts to write}
         READ      4 {DATO}   {It starts the data-read cycle}
         LOAD      4 {DATO}
         ADD=      10         {It computes the address of the cell in which
the
                              counter of the read-data is placed}
         STORE     3 {INDEX}  {It stores such address in INDEX}
         LOAD*     3 {INDEX}
         ADD=      1
         STORE*    3 {INDEX}  {A[INDEX] := A[INDEX] + 1}
         LOAD      2 {CONT}
         ADD=      1
         STORE     2 {CONT}   {CONT := CONT + 1}
         JUMP      CICLOL
SCRITT   LOAD=     0
         STORE     2 {CONT}   {CONT:= 0}
CICLOS   LOAD      2 {CONT}   {while CONT≤ K}
         SUB=      K
                   {write cycle}
         JGZ       FINE       {The execution is stopped at the end of the
cycle}
         LOAD      2 {CONT}
         ADD=      10
         STORE     3 {INDEX}
         LOAD*     3 {INDEX}
SCRC     JZFININD  {while A[INDEX] ≠ 0}
         WRITE     2 {CONT}   {write CONT}
         SUB=      1          {A[INDEX] := A[INDEX] - 1}
         JUMP      SCRC
FININD   LOAD      2 {CONT}
         ADD=      1
         STORE     2 {CONT}   {CONT := CONT + 1}
         JUMP      CICLOS
FINE     HALT
```

Solution of point b

Constant Cost Criterion

The program at point a. is composed by 3 sequential cycles. The first cycle (initialization) is executed *k+1* times. K being fixed and not depending on n, then it costs Θ(k) = Θ(1). The second cycle is executed *n* times; so it costs Θ(n). Finally, the third cycle is executed *k+1* times. At each iteration, however, it involves performing an inner loop: it writes k_i times the value *i* ($0 \leq i \leq k$). However, $\sum_{i=0..k} k_i = n$, and it also costs Θ(n).

Logarithmic Cost Criterion

Differently from the Constant Cost Criterion, it is necessary to consider the operation that depends from the value of *n*. These operations update the variable CONT and the value of the cells A[i], in fact, in the worst case, we could have that, for some index *i*, the value of A[i] is $\Theta(n)$. This kind of operations occur in the read and update cycle, and in the write cycle, where the cost is $\Theta(\sum_{i=0..n} \log(i)) = \Theta(n \log n)$.

Solution of point c

A Turing Machine with an arbitrary number of tapes could implement the proposed algorithm as follows.

The implementation we suggest uses a tape to represent each counter of the RAM (identified with A[INDEX] in the point a.). For simplicity, the counter uses an unary representation of numbers. The other tapes are used to store N and CONT, using the same meaning of the RAM program. Again, we use an unary representation. We suppose the data stored in the input tape and coded in the unary representation too.

Each single operation of the RAM program is simulated by the Turing Machine in the following way:

1. The read of *n* involves the computation of the entire input tape to the N tape: if *n* is coded in a unary representation, it costs $\Theta(n)$.
2. Each read of a single character, with the update of the related counter, could be done in constant time, since the data needs a limited number of cells; the selection of the counter to be update could be done by changing the state of the controller component during the read of the data; the update of the counter-tape involves the write of a single cell. Therefore, the entire read and update phase costs $\Theta(n)$.
3. The write phase needs the iteration of $\sum_{i=0..k} k_i = n$ times of a write operation of values included in the interval from 0 to *k*, costing $\Theta(1)$. The entire phase costs $\Theta(n)$.
4. The global computational complexity of the Turing Machine is $\Theta(n)$.

Solution of point d

A Turing machine with only 3 tapes can't use a tape for each counter. It would therefore be forced to store a number $\Theta(k)$ (at leas 2 if $K > 3$) of counters on the same tape. If we maintain the unary representation for the data, it involves that a single read and update operation costs $\Theta(n)$ instead of $\Theta(1)$. The global complexity will be $\Theta(n^2)$. If we use a binary representation, a single read and update operation costs $\Theta(\log(n))$, and the global complexity is $\Theta(n \cdot \log(n))$.

Exercise 3.1.7

Write a SMALG program that, given S as a finite set of cardinality equal to *n* composed by natural numbers and, at least, a natural number equal to *d*, verifies that if $\exists y_1, y_2 \in S \mid y_1 + y_2 = d$ and $y_1 \neq y_2$. Evaluate the complexity. Considering S as an ordered sequence, build an algorithm with linear complexity.

Solution

S could be represented as a vector *a*.

The easiest solution for this problem uses two nested cycles that verify that, for each couple of elements, their sum is equal to *d*. The time complexity of a similar solution is quadratic.

A more efficient solution uses a trick that usually reduces the complexity of the quadratic (or worst) algorithms: it orders the vector and, then, for each element *y* in the array, uses a binary search in order to search the value $y - d$. The time complexity of such solution, using an efficient sort algorithm, is Θ(n log n).

A more efficient solution could be obtained when the initial vector is already ordered. In such way it is possible to operate as follow:

Take the first element a[1] (the minimum) and the last element a[n] (the maximum). If a[1] + a[n] = d, we find the solution and the algorithm is finished; if a[1] + a[n] < d, then the minimum element value, added to any of the others, give as result a number smaller than *d*, because even if added to the maximum, result is smaller than *d*. In this case we could delete a[1] and search the couple in which the sum between the elements in position 2,…,n is *d*. If a[1] + a[n] > d, then a[n] could be eliminated because the sum of this number with anyone else gives as result a number bigger than *d*.

The algorithm continues eliminating, at each step, the minimum or the maximum of the left element. It stops when it finds the couple in which the sum is equal to *d* or when there are no more elements to examine. The time complexity is obviously linear because, at least, the algorithm computes exactly *n* iterations of the cycle.

The algorithm written in SMALG is:

```
var i,j,n,d,t;
begin
read d;
read n;
"leggi S in a[1] .. a[n]"
i:=1; j:=n; t:=0;
while (i<j) and (t = 0)
do
  if a[i] + a[j] < d
    then i:= i+1
    else if a[i] + a[j] > d
      then j:=j-1
      else t:=1
    fi
  fi
od
if t = 1
  then write i; write j
fi
end
```

Using the constant cost criterion (reasonable in this case), the complexity depends on the term k_1n+k_2 for each reading of the data in the memory and the initialization of the variables *i*, *t* and *j*, from the term k_3n for the cycle and k_4 for the write of the result. The complexity is $T(n) = k_1n+k_2+k_3n$, in other terms $\Theta(n)$.

Using the logarithmic cost criterion, the complexity increases of a logarithmic term: $\Theta(n \log n)$.

Exercise 3.1.8

Consider the following SMALG program, where some operations are described only in words, where M is a predetermined value:

```
begin
"Write 0 in a[1],..,a[M]"
read x;
while x ≠ 0 do
  a[x] := a[x] +1;
  read x;
od
"for each i, 1≤i≤M, print k times i if a[i] = k; don't
print i if a[i] =0"
end
```

The program reads as input a sequence of integers, terminated by a 0, and prints them in ascending order, using an array of counters.

Estimate its complexity. Check how the complexity changes in the case where M is not fixed, but it is a data input. Are the results in agreement with the known lower bounds on the complexity of sorting?

Solution

Following a uniform cost criterion, if n is the total number of data read, then the space complexity is $\Theta(1)$, while the time complexity is $\Theta(n)$. The fact that the complexity is linear is not surprising as since the lower bound for sorting ($\Theta(n \log n)$) is applies only to algorithms based on swap. The algorithm (similar to other linear sorting algorithms) is based on a particular assumption: M must be a fixed constant, that is the values of elements should be limited.

Even under this assumption, however, the earlier assessment obtained by the uniform cost criterion is not appropriate: if it happens, for example, that all the inputs elements are equal to a value i, then a[i] = n: if n can be very great need to use the logarithmic cost criterion.

In this case, the increase of the counter costs at most log n and then it gives the classic result $\Theta(n \log n)$ time complexity. Even the spatial complexity is changed: S(n) is $\Theta(\log n)$.

Note that, under the same assumptions and adopting the same criterion of cost for the Quicksort you get more a time complexity $\Theta(n \log n)$, but a spatial complexity $\Theta(n)$.

A particular case of interest in applications, where the assumption of uniform cost is reasonable, occurs when M is prefixed and the n input values are certainly all distinct. In this case, the counters may contain only 1 or 0 and you can use a single bit. The data structure becomes a BITSET, that is an array of bits.
For example, to sort natural numbers of 16-bits, all distinct, it needs 65536 bits, that is just over 8 Kbytes of memory.
The resulting complexity of the algorithm is linear, and also the constants are very small, compared to the classical algorithms for exchanging, enabling very fast sorting.
Suppose now that M is not fixed, but depends on the input, and that there is no other assumptions on the data values.
Using a logarithmic cost criterion, we can make an estimate of the algorithm of the counters with the dominant operator, a[x]:=a[x]+1.
The cost of this operation is, in the worst case, $\log n + \log M$. Therefore $T(n)$ is $\Theta(n(\log(n)+\log(M)))$.
For example, if $M \in \Theta(n)$ then the resulting complexity is still $\Theta(n \log n)$. If instead $M \in \Theta(2^n)$ then the complexity is $\Theta(n^2)$.

Exercise 3.1.9

On the language below:

$L = \{ab^{n1}ab^{n2}...ab^{nk} \mid k \geq 0, n_i > 0 \text{ per } i = 1, ...k, \exists i, j, 1 \leq i < j \leq k, n_i = n_j\}$

- Build a Pushdown Automaton or a Turing Machine that identify the language L. Create a RAM machine that identify language L.
- Evaluate the complexity of recognizing language L using one of the abstract machines created before (part a of the exercise). Assume constant cost for the second part of the exercise.
- (*) Evaluate the complexity of the recognition of language L using a RAM machine, with logarithmic cost. Also solutions with a complexity degree which stands below the complexity of L are assumed correct.
- (*)Discuss whether it is possible to obtain a better complexity than those obtained with the machines discussed in point a, and if so briefly explain how.

Solution of point a

The easiest way is to use a Non Deterministic Pushdown Automaton, which chooses, in a non-deterministic way, a group of b^{ni} to be inserted on the Pushdown Automaton. As before, another group of b^{ni} is used in order to be compared with the first one, deleting characters from the Pushdown Automaton.

A RAM machines which is able to identify L, works as described below:

1. To any group b^{ni} is assigned a counter, which is incremented by one every time a b is read during scanning of the group b^{ni}.

2. K counters are compared one to each other, with two nested cycle: the extern cycle changes the j index from 1 to k-1; the inner cycle change the i-index from j+1 to k. Obviously as you found a couple of index with equal counter you could stop the execution of the cycles.

Solution of point b

The complexity of the recognition of L language, through a Non Deterministic Pushdown Automaton is $\Theta(n)$.

A RAM machine, which is able to work as described in section a.2, requires constant cost criterion.

1. $\Theta(n)$ for the first part.
2. $\Theta(k^2)$ for the second part.

It is required to evaluate the relationships between k and n (in the worst case). It is not enough to verify that k could be $\Theta(n)$, because in that case the value of the counter could be small, and the extern cycle could not be executed $\Theta(n)$ times. By the way, we can assume as upper bound of k the n value, and state that the complexity is (probably overestimated) $\Theta(n^2)$.

A similar situation which is close enough to the worst case is given with a certain input: $ab^cab^{c-1}ab^{c-2}$ …ababab…abab, with $c = \Theta(\sqrt{n})$ and the number of couples ab equal to $\Theta(n)$. In that case the extern cycle will be executed $\Theta(\sqrt{n})$ times, and for each iteration the inner cycle is executed $\Theta(n)$ times. The overall complexity is equal to $\Theta(n^{3/2})$.

Solution of point c

Constant Cost Criterion Analysis is more complex.

- The first part is$\Theta(n.\log(n_h))$, where $n_h$is the maximum of the counters.
- The second part$\Theta(k^{2\cdot}\log(n_h))$.

We should evaluate values assumed by k and n_h in the worst case, (remember that the two values are not independent one with each other). With input equal to ab^{n-1}, k è $\Theta(1)$ and n_h è $\Theta(n)$, the complexity is $\Theta(n\cdot\log(n))$. But in the case analyzed in part b would have a complexity of $\Theta(n^{3/2\cdot}\log(n))$. NB: When k is $\Theta(n)$, n_h is small, and probably exit early from the extern cycle.

Instead of going further in a more detailed analysis of the overall complexity of the values assumed by k and n_h depending on n, we could just use an, easy obtained, overestimation obtained assuming, at the same time, the worst values of both k and n_h, then $\Theta(n)$ for each one. It produces a total complexity (probably overestimated) equal to $\Theta(n^{2\cdot}\log(n))$.

Solution of point d

Obviously, it is not possible to obtain a complexity less than $\Theta(n)$. It is obtained using a non deterministic machine. We would have a deterministic machine, so it is easy to build a deterministic machine that recognize the language L in $\Theta(n^2)$: the different groups b^{ni} are stored in a tape and compared, one by one, with the remaining part of the string. It is not easy to obtain a complexity that is less than this limit. The authors – and probably the state of art – do not know the lower bound of recognizing the language L (in a deterministic way).

Exercise 3.1.10

Evaluate the complexity of the following recursive algorithms written in SMALG$^+$:

1.

```
procedure fact(n)
begin
     if n= 0 then return 1 else n * fact(n-1) fi
end
```

2.

```
proc order(n) /* in a[0] is stored the minimum */
begin
if n>1 then     order(n-1);
                while a[n] < a[i] do i:= i-1 od
                if i≥1 then x:= a[i]; a[i] := a[n]; a[n] := x; fi
fi
end
```

3.

```
begin
if s<d then
        m := (s+d) div 2;
         order(s, m);
         order(m+1, d);
         i := s; j:= m; k :=0;
         while i ≤ m-1 or j ≤d do
             k := k+1;
             if a[i] ≤ a[j] or j >d then t[k] := a[i]; i := i+1;
                                  else t[k] := a[j]; j := j+1; fi
         od
         while k≥i do
             a[k] := t[k];
             k := k -1;
         od
fi
end
begin
        "read n elements in a"
        order(1, n);
end
```

Solution

1. Supposing that the cost to execute the procedure fact with n=0 is k_1, and the cost of the test of the else branch is k_2, the time complexity is given by:

$$\begin{cases} T(n) = k_2 + T(n-1) \\ T(0) = k_1 \end{cases}$$

This kind of equations is defined as *finite recurrence equation*. The expression for T(n) is in a *closed form* if it satisfy the equations and do not contain the function T. In this case is $T(n) = k_1 + k_2 n$.

2. With the uniform cost criterion, in the worst case, a[n] must be inserted in the first position at each call of the procedure (the array was ordered in a decreasing order). The resulting equation could be written as:

$$\begin{cases} T(n) = c_1 + T(n-1) + c_2 n \\ T(0) = c_3 \end{cases}$$

Where c_1,c_2 and c_3 are appropriate constants. T(n) is obviously $\Theta(n^2)$ because, if it was T(n) = T(n-1) +n with T(1)=1, then T(n) = n*(n+1)/2. If we want to compute also the coefficients, we could substitute the quadratic polynomial $an^2 + bn + c$ in the previous equations and, using the principle of polynomial identities, obtain the expressions in the coefficient a, b and c.

$$\begin{cases} a\ n^2 + b\ n + c = c_1 + a(n-1)^2 + b(n-1) + c + c_2 n \\ c = c_3 \end{cases}$$

From the first equation we get: $0 = c_1 - 2na + a - b + c_2 n$, cioè $a = {}^{c_2}\!/_2, b = c_1 + {}^{c_2}\!/_2$, $c = c_3$.

3. This algorithm is a version of the mergesort.

The cost to execute order(1,n) is T(n) = k1 + 2T(n/2) + k2*n, where *k1* e *k2* are appropriate constants. The following theorem is valid:

The solution of the equation $T(n) = aT(n/b) + cn^k$, where *a* and *b* are integer constants, $a \geq 1$, $b \geq 2$, and *c* and *k* are positive constants, is:

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & \text{se } a > b^k \\ \Theta\left(n^k \log n\right) & \text{se } a = b^k \\ \Theta\left(n^k\right) & \text{se } a < b^k \end{cases}$$

Then, considering k =1 and a =b, the complexity is $\Theta(n \log n)$.

Exercise 3.1.11

1. Formalize the model of a Turing Machine with two-dimensional tape (the tape could occupy the entire Cartesian Plane or just one quadrant)
2. Define a relationship between the complexity of the machine formalized at point 1 and the traditional k-tape Turing Machine.

Solution of point 1

The most important point of the formalization consists in the representation of the tape configuration and its transformation using the transition relation. The traditional approach, that considers the tape as a string of characters, gives the following solution:

1. Consider that the tape exists in the first quadrant of the integer Cartesian plane.
2. Using a string, build a square with a vertex in the origin of the plane that include the set of the tape containing the non-empty cells
3. Consider as 1 the size of one side of the square
4. The content of the tape is formalized by the string $a_{11}a_{12}\ldots a_{11}\$a_{21}a_{22}\ldots a_{21}\$a_{r1}a_{r2}\ldots a_{ri}{*}a_{ri+1}\ldots a_{r1}\ldots\$a_{11}a_{12}\ldots a_{11}$
   where * represents the position of the head
5. The transition from a configuration to another one of the tape is governed by the following rules:

   ← a11a12...a1l\$a21a22...a2l\$...ar1ar2...ari*ari+1......arl ... \$ al1al2...all ⊢
   a11a12...a1l\$a21a22...a2l\$ ar1ar2...ariari+1*......arl ... \$ al1al2...all

if the value of the δ function specifies the movement of the head to right

← a11a12...a1l\$a21a22...a2l\$ ar1ar2...ari*ari+1......arl ... \$ al1al2...all ⊢ a11a12...a1l\$a21a22...a2l\$...ar-1,1ar-1,2...ar-1,i*ar-1,i+1......arl\$ar1ar2...ari*ari+1......arl ... \$...al1al2...all
if the value of the δ function specifies the movement of the head to the bottom

…

In particular

← a11a12...a1l\$a21a22...a2l\$...ar1ar2...*arl ... \$ al1al2...all ⊢ a11a12...a1la1,l+1\$a21a22...a2la2,l+1\$...ar1ar2....arl*ar,l+1... \$ al+1,1al+1,2...al+1,l+1
with as, l+1 = al+1,s = blank for each s, if the value of the δ function specifies the movement of the head to right

Solution of point 2.a

Assuming that dimensional Turing Machine M_M simulates the two-dimensional machine M_B, storing the contents of the tape of M_B as a string built as defined in the point 1. In general, the simulation of a single movement of M_B force by M_M takes a time proportional to the tape length. This is equal to the area of the square, equal to l^2. In the worst case, for each movement of M_B, l is increased by 1. The length of the tape of M_M is $\Theta(T^2(n))$. At the end, the entire simulation takes $\Theta(T^3(n))$.

Solution of point 2.b

Assuming that a two-dimensional Turing Machine M_B simulates a dimensional M_M with k tapes, storing the contents of the various tapes in k rows, one on each other (or, in the same way, on the same row, one after each other). In this case, the M_B, having a single head, should track, using appropriate symbols, the position of the simulated heads. Tracking this positions and simulates the changes determined by a single movement of M_M will take, in general, a time proportional to T(n). Consequently, the entire simulation will take $\Theta(T^2(n))$.

Exercise 3.1.12

1. Formalize the model of a queue automaton that works as follows:
   - ← The automaton reads sequentially the characters from an input tape;
   - ← The movement is summarized as follow: read the character from the input tape, one from the tail and one from the current state; than change the current state and write a finite string at the end of the queue;
   - ← The automaton could executes ε-movements both on the input tape and the memory tape;
   - ← The recognition is based on the fact the automaton is in an accepting state after the reading of the entire input string

If the above definition in not precise, the student could highlight the ambiguities and choose the best way to formalize the model, taking care to giving convenient motivations (for instance, considering the "realism" of the model)

2. Define (justifying the answer) if the model of the queue automaton, formalized at the point 1, has the same computational power (recognizes the same class of languages) as the Turing Machine or not.
3. Define the relationship between the queue automaton, formalized at the point 1, and the traditional k-tape Turing Machine.

Solution of point a

The queue automaton (deterministic recognizer) AC is defined by 7 parameters $<Q, I, \Gamma, \delta, q_0, Z_0, F>$ where the various symbols have the traditional meaning of: state set, input set, queue symbol set, transition function, initial state, initial state symbol, acceptance state set;

$\delta:(Q \times (I \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})) \rightarrow Q \times \Gamma^*$

Similarly to the analogous definition of pushdown automaton, in order to have AC as a deterministic machine, we impose that, if $\delta(q, \varepsilon, C)$ is defined (for each q, C), then $\delta(q, i, C)$ could not be defined for any $i \in I$; if $\delta(q, f, \varepsilon)$ is defined (for each q,f), then $\delta(q, f, A)$ could not be defined for any $A \in \Gamma$.

A configuration of AC is a set of $c = <q, x, \alpha>$, $x \in I^*$, $\alpha \in \Gamma^*$.

The transition relationship between the configurations is defined as follows:

$c = <q, x, \alpha> \vdash c' = <q', x', \alpha'>$ if and only if:

- $x = ix'$; $\alpha = A\beta$, $\alpha' = \beta\gamma$ e $\delta(q, i, A) = <q', \gamma>$
- $x = x'$; $\alpha = A\beta$, $\alpha' = \beta\gamma$ e $\delta(q, \varepsilon, A) = <q', \gamma>$
- ....

The string $x \in I^*$ is accepted from AC if and only if $<q_0, x, Z_0> \overset{*}{\vdash} <q, \varepsilon, \gamma>$ with $q \in F$.

Note that, in this way, the concept of ε-movement is formalized in order to keep the tape unchanged by AC (input tape or the head of the queue) and also without dependencies with its first cell.

An alternative formalization could have been used to link the first movement with the first symbol of the given tape (input or queue) allowing the machine to avoid any kind of movement of the head. In this case, the domain of δ does not require the symbol ε; in contrast to this, the co-domain needs an instruction that specifies if the head is moving or not, analogously to what happens in a Turing Machine. It easy to show the computational equivalence of the two types of formalism: to simulate the second with the first, it is enough to read a symbol and maintain in the memory, using the finite states of the control component; the opposite simulation is simpler because the finality of the first machine could be shown as a particular kind of the second one.

Solution of point b

An AC could simulate the behavior of a TM in the following way:

The queue of the AC stores the contents of the tape of a single tape TM (initially, AC copies the whole contents of its input tape in the queue, and then starts the simulation of the TM). An appropriate symbol indicates the position of the head of the TM, as well as its state.

To simulate the single movement of the TM, AC executes a complete "rotation" of the queue. During the rotation, AC keep track, using the finite state memory of the last read symbol, until it finds the mark that indicates the head (and the state) of the TM. At this point it could simulate the movement of the TM storing a symbol string in the queue that contains a part of the tape that has been modified by the TM movement.

For instance, suppose that the queue of the AC contains the following string:

α DAqBC β

where the symbol q represents that the head of the TM is placed at the position of the character B and the state is q.

Suppose that the movement of the TM rewrites B' instead of B, move itself to q' and move the head to the right.

To simulate the movement, AC scans and rewrites the queue until D (included), after reading and storing A using an appropriate state. The current configuration is:

qBC β α D

At this point, the machine reads q and B modifying the tape in:

C β α D

Now AC has the information needed to accomplish the movement and to rewrite in the queue the string qAB'. The final configuration is:

C β α DqAB'

At the end, the machine completes the rotation of the queue in order to reach the following situation:

α DqAB'C β

(Clearly, in order to facilitate the operations, the appropriate symbols could mark the start and the end of the tape).

Solution of point c

The solution proposed above (for the point b) shows that the AC, to simulate a single movement of a TM, must scan the entire queue, having a length equal to the length of the tape. Hence, if the TM has a complexity equal to T(n), the AC, obtained with the previous solution, has a complexity equal to $T^2(n)$.

In contrast, suppose that you want to simulate an AC using a TM. It is natural to encode the queue of the AC using a tape of the TM. However, in order to simulate a single movement of the AC, the TM must access both the head of the queue (the rightmost part of its tape) to reading and the end of the queue (the leftmost part of the tape) to writing. Since every tape has a single head, also in this case the simulation of the single movement requires a whole scan of the tape. In this case if the AC has a complexity of T(n), the TM, obtained as described above, has a complexity of $T^2(n)$.

Exercise 3.1.13

Consider the following language:

$L = \{w^k | \, k \geq 2, w \in \{a, b\}^*\}$

(strings of L are: aa, abaabaaba, abbabbabbabb, ...}

1. Build (not necessarily in detail) a Turing machine that recognizes L.
2. Build a RAM machine that recognizes L.
3. Evaluate the complexity of the recognition of L through abstract machines built in paragraphs a. and b. In case of RAM machine, use the logarithmic cost criterion.
4. Explains how to solve the same problems mentioned in paragraphs a., b., c. replacing the L-language with L' defined as L but with $k \geq 1$.

Solution of point 1

Simple TM that recognizes L operates as follows:

3. For increasing values of k (from 2 and the most up to n, n = |x|):
    1. It divides by a first pass (through appropriate markers) x in k substrings w_i of length n/k (if n is not a multiple of k, the pass failures and move the following value of k).
    2. Check whether the various substrings w_i are all equal.

Solution of point 2

At the first instance a RAM recognizing L could operate in the same pattern as the TM. You only need to store in the first place the input tape in memory (using a cell for each character) because the input instructions of the RAM do not allow to go back with the read head after performing a READ.

It also be better, as for the TM, using different areas of memory (different tapes for TM, sequences or arrays of cells attaching to the RAM) to make scanning easier and more efficient that decomposes x into substrings.

Solution of point 3

The complexity of the TM in paragraph a. is clearly $\Theta(n^2)$. In fact, for every k, it takes two full scans of the input string (its division into substrings it doubles at most as the length; so the number of moves performed for each k is still $\Theta(n)$).

The complexity of the RAM that executes the same algorithm is also $\Theta(n^2)$, if we use the constant cost criterion. Otherwise, using the logarithmic cost criterion it is necessary to keep in mind that the access to the n-th cell of the machine costs log(n). So each pass costs $\Theta(n \log(n))$ (remember that $\Theta(\sum_{i=1}^{n} \log(i)) = \Theta(n \log(n))$); the total complexity is $\Theta(n^2 \log(n))$.

Observation

A more sophisticated algorithm could be obtained, for the TM and the RAM, observering that k must be a divisor of n. Therefore, if $x = w^k$, with $|x| = n$, $|w| = n/k$, and $h = k/r$, then also $x = y^h$, with $y = w^r$. Consequently, advance decomposing n in a product of powers of prime number, it is enough to apply a verification of the value of k that are equal to such values. It is not easy, nevertheless, to estimate, in the worst case, given a number n, how many prime factors are contained in it.

Solution of point 4

If $k \geq 1$ it is obviously possible to apply the same algorithm with the appropriate changes. Observe that, in this case, L' = {a, b}* and then it could be recognized using an easy algorithm.

Exercise 3.1.14

Please write a RAM program that computes the function $f(n) = 2^{2^n}$ and estimate the complexity by the constant cost criterion and the logarithmic cost criterion .
(*) Solve the exercise without using the following instructions: RAM MULT and DIV

Solution of point a

An easy RAM program that calculates the function f(n) is:

```
             READ          1 {N}
             LOAD=         2
             STORE         2 {RIS}     {RIS := 2}
             LOAD=         1
             STORE         3 {COUNT}        {COUNT := 1}
LOOP         LOAD          3 {COUNT}        {WHILE COUNT ≤ N}
             SUB           1 {N}
             JLZ           END
             LOAD          2 {RIS}
             MULT          2 {RIS}
             STORE         2 {RIS}     {RIS := RIS²}
             LOAD          3 {COUNT}
             ADD=          1
             STORE         3 {COUNT}        {COUNT  :=  COUNT  +
1}
             JUMP LOOP
FINE         WRITE         2 {RIS}
             HALT
```

The complexity of this program by the constant cost criterion is $\Theta(n)$.

By the logarithmic cost criterion, the complexity of the operations inside the loop is a function of the value stored in the RIS memory cell that has the values 2^{2^i} in sequence, per i=1,…, n. The cost of the access, updating and multiplication operations in that cell is $\Theta(\log(2^{2^i})) = 2^i$ for the i-th iteration of the loop. The total complexity of this program is $\Theta(\sum_{i=1}^{n} 2^i) = \Theta(2^{n+1})$.

Solution of point b

The easiest way to solve this problem is to substitute the MULT operation with a loop of sums, so you can have a single multiplication that costs 2^i instead of $2^i.2^{2^i}$!

You can have better results by keeping on mind that an integer is representable in the binary numeration as a string of bit and also that the multiplication 2 by m is achieved by adding a 0 in the bottom of the string of bits that encodes m.

Therefore we can write an algorithm that uses this principle by referring to a TM (in this case a RAM is useless because is not provided of mechanisms that enables to access to data representation: so it should simulate the TM by representing in each cell the numbers 0 and 1 for building the representation of m).

The binary value 2^{2^n} is the number 1 followed by 2^n zeros. Thus we need to build a string of 2^n zeros, given n. A TM can do this in the following manner:

- storing the unary encoding of n in the tape N1. If n would be given in a non unary encoding, its conversion would take a time that not changes the order of size of the whole complexity computation.
- storing 1 (= 2^0) 0 symbol, always in the unary form in another tape,
- doing n runs where each one doubles the content of the tape N2. More precisely, the content of N2 is doubled and stored in N3; in the next run the content of N3 is doubled and stored in N2 and so on: as a result, after the i-th running, with odd, will have 2^i zeros in N3, after the i-th running, with even, will have 2^i zeros in N2.
- Finally, putting a 1 in front of produced zeros would be enough.

Clearly, without taking count of the initialization that will require a tiny time anyway, each running will take a time $\Theta(2^i)$ and then the total time will be $\Theta(\sum_{i=1}^{n} 2^i) = \Theta(2^{n+1})$, as in the RAM case with a MULT as elementary operation

Exercise 3.1.15

Consider a single-tape Touring machine model with many different read-write heads that can move independently of one another.

Study the existing complexity relation between this model and the traditional k-tape single head machine.

Solution

In general, the reduction of the number of a TM tapes to one, involves a complexity $T^2(n)$, because the simulation of a single move in the original machine needs a number of move that go thorugh, in the worst case, the entire memory, which order of size is $\Theta(T(n))$. Notice that in some case the multiplicity of heads can substitute the multiplicity of tapes: you can, indeed, allocate different portions of a tape to the simulation of the original machine single-tape. In the worst case, however, limited portions of a tape, would not be enough to contain all the information of the original machine tape; this will force the shift of unlimited portions of tape, obviously the order of size is T(n). The opposite simulation (a k-tape TM simulate a 1-tape TM with n-heads) also involves a complexity $T^2(n)$. Also in this case, indeed, the k-tape TM will perform a number of move of order T(n) to access to the information placed in correspondence to the various tapes in which, instead, the original machine can access by a single move.

Exercise 3.1.16

Analyze the complexity of the machine built in Exercise 1.3.3 (whatever formal model you adopt, complexity is calculated as the number of elementary moves performed by the machine to operate the recognition of a string). If it is not linear with respect to the length of the string in input, try to modify the machine so as to obtain a linear complexity.

Solution

The machine described in the solution of Exercise 1.3.3 has linear complexity. Indeed, said *m* the length that will take, in the worst case, 3m + k steps. The length *n* of the whole string, when it belongs to L, is 2m + 1. Taking into account the fact that, as always, we have the worst case for the complexity when the input string belongs to the language (we can stop the analysis when we are sure the input string don't belongs to L), linearity of the complexity of this type of recognition is clear.

Exercise 3.1.17

Solve Exercise 3.1.16 trying to obtain real-time complexity, such that $f_M(n) = n$, f_M is the complexity function of the machine M. Pay attention: it is not enough that $f_M(n) = \Theta(n)$.

Solution

The recognition made by the machine of the solution of Exercise 1.3.3 is not real-time, but it is linear. Not even in this case the linear speedup theorem can help in solving the problem (it requires that the original complexity of the machine to be speeded up in real-time is higher, considering a negligible ε arbitrarily small).

The most natural way to recognize L in real-time is to build an ad-hoc abstract machine; it is provided of a memory structure studied expressly to make this recognition. In this way it stands to reason that the abstract machine suitable for the problem has a double memory structure: a stack and a queue. Notice that, differently from the Turing machine previously used, the “stack” tape behaves in the same way of a structure. This not happens for the “queue” tape that requires a repositioning of the head after reading ‘c’.

It is necessary to define a machine in which the stack has the traditional “push” and “pop” operations, respectively to and from the queue; the queue has an enqueueing operation and an extraction operation from the front of the queue.

The machine configuration will be described from a quadruple $c = \langle q, x, \pi, \gamma \rangle$, where π shows the content of the stack and γ the content of the queue.

Assumed $c_1 = \langle q_1, iy, A\pi, B\gamma \rangle$, the machine will go from c1 to $c_2 = \langle q_2, y, \alpha\pi, \gamma\beta \rangle$ if and only if

$\delta(q_1, i, A, B) = \langle q_2, \alpha, \beta)$.

The completion of the definition (definition of δ, handling of steps that do not involve heads movements, acceptance …) is left to the reader.

Now it is very simple to use such a machine in order to recognize the language L of the beginning. It is also clear that this machine will work exactly as the Turing machine used at the beginning; it will not require any step to reposition the head of the queue: it will put read symbols in the queue until the reading of ‘c’. At this point the machine will end to put symbols in the queue and will start to take symbols from the queue in order to compare them with characters read.

In this way all the steps will read a character from the input tape, so we will execute exactly n steps, where n is the length of the input string.

Notice that we could have same results using a Turing machine with more tapes and more heads: in this way a head of the queue tape represents the front of the queue and another head the bottom.

Exercise 3.1.18

Does an algorithm with a linear complexity have complexity of $\Theta(n)$? Justifying the answer.

Solution

NO: the function $f(n) = n + \log(n)$ has a complexity equal to $\Theta(n)$, but is not linear. Instead, it is true that a linear function has always a complexity of $\Theta(n)$.

Exercise 3.1.19

Identify and justifying which sentences are true, and which are false.

1. A RAM machine could solve every NP-complete problem with a polynomial number of cells, with respect to the size of input data, using logarithmic cost criterion.
2. A RAM machine could solve every NP-complete problem in an exponential time with respect to the size of input data, using logarithmic cost criterion.
3. If a RAM machine, without MULT and DIV instruction, has a polynomial spatial complexity using the constant cost criterion, then it has a polynomial spatial complexity using logarithmic cost criterion.

Solution of points a e b

Both the spatial and time complexities are polynomially related to the corresponding complexities of the Turing Machine if logarithmic cost criterion is used (it is easy to extend the theorem of the polynomial correlation to the spatial complexity). The property a. and b. are true for the Turing Machine: in fact, a) NP $\subseteq$ PSPACE; b) if the non-deterministic machine has a complexity equal to f(n), there exist an equivalent deterministic machine with a complexity equal to $c^{f(n)}$. The sentences are true also for a RAM with logarithmic cost criterion.

Solution of point c

No. In fact, in a single cell of a machine, using an appropriate sequence of operations, it could be possible to store an arbitrarily bigger value (for instance 2^{2^n}). Its storing, using logarithmic cost criterion, is equal to the logarithm of its value (in our example is 2^n), while, using constant cost criterion, it costs 1.

Exercise 3.1.20

Identify and justify which sentences are true, and which are false.

- Consider two monotonic positive functions f_1 and f_2, defined on the domain of natural numbers. $f_1(n) > f_2(n)$ for each n implies that DTIME(f_1^2) $\supseteq$ DTIME(f_2^2).
- The cardinality of the difference set (DTIME(n^2) - DTIME(n) is strictly greater than the cardinality of the set DTIME(n).

Solution of point a

NO: if, for instance, $f1(n) = 20n^2$ and $f2(n) = 10n^2$, $f1^2(n) = 4f2^2(n)$ but, using the theorem of linear speedup, DTIME($f1^2$) = DTIME($f2^2$).

Solution of point b

NO: both sets have cardinality equal to $\aleph_0$, that is the cardinality of the countable sets. In fact, both are subsets of decidable problems (that are countable), and both are infinite.

Exercise 3.1.21

Consider the so called *gang problem* (GP), thus defined: given an undirected graph G=(V,E) determine the largest complete sub-graph of G. A sub-graph is *complete* if every pair of vertex is connected by an edge. Show that the problem is NP-complete.

Solution

We define the GP problem in this way: given a graph G and an integer k, verify if there exists a complete sub-graph composed of at least k vertex. The resolution of GP for a generic k allows to find quickly the maximum value of k (as $k \leq |N|$) and to solve the gang problem[4]. Then, it is sufficient to demonstrate that GP is NP-complete.

GP is in NP, because an NTM can solve the problem in polynomial time: the NTM chooses, in a non-deterministic way, a k order sub-graph of G and verifies if the sub-graph is complete (it is sufficient to chek each vertex and if there are the edges connecting it with the other vertexes).

We demonstrate that GP is NP-hard by reducing the satisfiability problem SAT to GP in polynomial time.

Given a formula in conjunctive normal form with k clauses, we build a graph G by matching a node at every occurrence of variables. Moreover we connect an edge between each pair of nodes, with exception of the following cases:

3. two vertexes correspond to variable occurrences of the same clause
4. a vertex corresponds to the denied occurrence of the same variable which refer to the other vertex

For instance for the formula $(x1) \wedge (x'1 \vee x2 \vee x3) \wedge (x'3)$ we obtain the graph shown below.

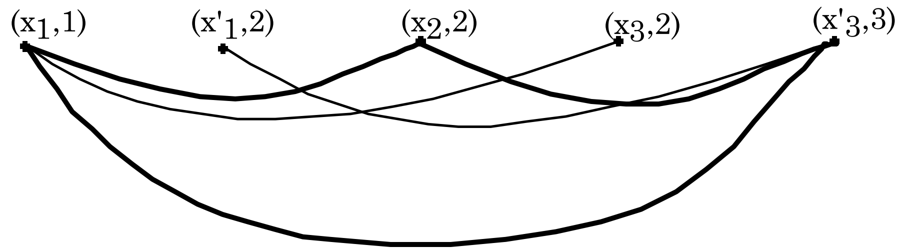


Figure 3-7 The complete sub-graph is marked with a heavy stroke.

The CS problem is formally built starting from a formula F by defining a graph G = (N,A) and an integer value k as follows:

$N := \{(x,i) : x \in \text{i-th clause}\}$

$A := \{ \{(x,i), (y,j)\} : x \neq y', i \neq j\}$

K:= number of clauses

The building of the graph has a polynomial complexity with respect to the number of variables of the formula.

Now we prove that the graph defined above has a spatial complexity of order k if and only if the formula is satisfiable.

Considering the condition (1), if a complete sub-graph of k vertexes exists, considering that each vertex could contains at least one clause and the clauses are k, the sub-graph has one and only one vertex for each clause. Assign to the correspondent variables of the CS vertexes true if the occurrence is not complemented, false if it is complemented, in other terms, xj = true if $(xj, i) \in CS$, false if $(xj', i) \in CS$. About the condition (2), the assignment is not contradictory and, since every clause is true, the formula is true. Vice versa, if the formula is satisfiable, then it exists an assignment σ of truth values that makes every clause true, then, for each vertex group that corresponds to the same clause, we select an occurrence that makes true the clause with the assignment σ. Since the (1) and the (2) do not exclude edges between the selected vertexes. The resulting sub-graph is complete and k-order.

For instance, the complete sub-graph marked in figure corresponds to the assignment x_1 = T, x_2 = T, x_3 = F that verifies the given formula.

Exercise 3.1.22

Consider the know problems of the *node coverage* (NC) of a non oriented graph, defined as: given a non oriented graph G = (V,E), and an integer number k, verify the existence of k vertexes in which each edge of G is connected to, at least, one of the k vertexes, or, to put it better, if exists $S \subseteq N$ such that $|S| \leq k$ and $\forall a \in A$ $a \cap S \neq \emptyset$. Prove that NC is NP-complete.

Solution

NC is in NP (it is not difficult to build a NTM able to solve the problem in a polynomial time).

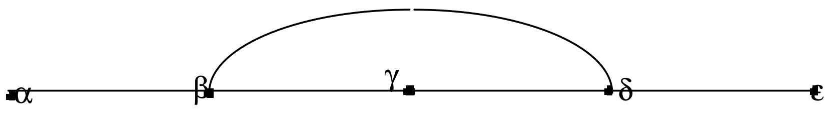
We prove that NC is NP-hard simplifying the problem of the complete sub-graph (CS) to NC in polynomial time. Given a graph G = (N,A), build a graph G' (without self-loop) with the same vertexes of N, and with all and the only that do not belongs to G:

N':=N

A' := { (i,j} : i≠j, (i,j) ∉ A}

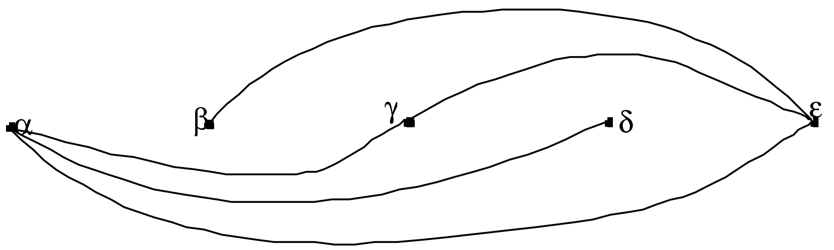
k':= |N| - k

Have a look at the example of Figure 3-8 and Figure 3-9.



G=(V,E) ; k=3

Figure 3-8: A graph with 5 vertexes and 5 edges.



G'=(V',E') ; k'=2

Figure 3-9: Graph G' obtained from G by complement.

If it is possible to cover G' using vertexes (in the example, using α and ε), so it exists a complete sub-graph of G; it is composed of vertexes that not belong to the coverage (in the example β, γ, δ) and the edges that connect them. In fact, the vertexes that not belong to the coverage could not be connected, in G', using edges (otherwise such vertex will avoid the coverage): each vertex that not belongs to the coverage and then it is, for definition of G and G', connected to an edge to all the other vertexes that not belong to the coverage: the set of each vertexes forms a complete sub-graph of G

Exercise 3.1.23

Tell, justifying your answer, if it is better, by time complexity point of view, to use a RAM (applying logarithmic cost criterion) or a Turing Machine to recognize $L = \{a^n b^{n^n} | \ n \geq 1\}$.

Solution

Let m be the length of the string to read: $m = n + n^n$, so m is $\Theta(n^n)$.

L can be recognize by Turing Machine M with 4 tapes, and in tape 3 there is the partial result of n^i elaboration, tape 1 is a counter from 1 to n, its head is in i-th position, and tape 2 contains the value of n, tape 4 is used like a support memory. The machine can work in the following way:

1. Read a^n and store n in unary on all of its tapes.
2. Multiply the content of tape 3 by n and store the result on tape 3. After this, it counts "1" shifting head of tape 1 by one position. To execute multiplication of the number contained in tape 3 by n it uses tape 2 and tape 3 and it executes n sums of the result of tape 3 with itself using tape 4. This operation requires $\Theta(n^2)$ steps.
3. Repeat the multiplication of result of tape 3 by n (still contained in tape 2) for other n times, each time shifting by one position the head of tape 1: when tape 1 is completely read, tape 3 will contain n^n unary codified. Each multiplication requires a number of steps proportional to the given result, so it is $\Theta(n^i)$. To calculate n^n will be necessary a number of steps equals to

$\sum_{i=1}^{n} n^i$ che è $\Theta(n^{n+1}) = \Theta(n \cdot n^n)$

4. Now, it reads "b" characters comparing with the unary number contained in tape 3: the

string will be accepted if the two numbers are equals. This comparison requires n^n steps.

In conclusion, the entire computation has $\Theta(n \cdot n^n)$ time complexity.

Instead RAM must reads and counts all the "a" and the "b". A counter that stores n^n requires a space of $\log(n^n)$ (that is $\Theta(n.\log(n))$ cells of memory).So the entire reading and the counting require time at least $\Theta(m.n.\log(n)) = \Theta(n^n.n.\log(n))$. The total complexity of a RAM is worst then M.

Exercise 3.1.24

The following RAM program computes the sum of the squares of the first *n* integers (*n* is the input), computing the square of a number *k* as the sum of the first *k* odd numbers.

```
            READ      1    {M[1] = n}
            LOAD=     0
            STORE     2    {M[2] = 0}
            LOAD=     1
            STORE     3    {M[3] = 1}
OL:         LOAD=     0
            STORE     4    {M[4] = 0}
            LOAD=     0
            STORE     5    {M[5] = 0}
IL:         MULT=     2
            ADD=      1
            ADD       4
            STORE     4    {M[4] = M[4]+M[5]*2+1}
            LOAD      5
            ADD=      1
            STORE     5    {M[5] = M[5]+1}
            SUB       3
            JZ        AD   {if M[5] = M[3] goto AD}
            LOAD      5    {M[0] = M[5]}
            JUMP      IL
AD:         LOAD      2
            ADD       4
            STORE     2    {M[2] = M[2]+M[4]}
            LOAD      3
            ADD=      1
            STORE     3    {M[3] = M[3]+1}
            SUB       1
            SUB=      1
            JZ        WR   {if M[3]-M[1] = 1 goto WR}
            JUMP      OL
WR:         WRITE     2
            HALT
```

Please state the complexity class of the program, referring both to uniform and logarithmic cost criteria.

Solution

The given program implements the algorithm shown in Figure. The variables have the following meaning:

1. m1 stores the value of n;
2. m2 stores the result;
3. m3 is a counter from 1 to *n*;
4. m4 in used to store the square of m3;

m5 is used to scan the odd numbers.

Figura 3-7 Program that computes the sum of the squares of the first *n* integers.

It is then easy to state that the outer cycle is executed *n* times (every time m3's value is increased). For every iteration of the outer cycle, the inner one is executed m3 times.

With respect to uniform criterion complexity is then $\Theta(n^2)$.

With respect to logarithmic criterion, keeping in mind that the main term is the one related with the construction of a new square in m4 in the inner cycle and that the space occupied by m4 is $\Theta(n^2)$, complexity class is $\Theta(n^2 \log n)$.

With logarithmic criterion spatial complexity is determined by m2, that, being m2 the sum of the squares of the first *n* integers, is $\Theta(n^3)$.

Exercise 3.1.25

Please build (even not so detailed but sufficiently precise) a bi-dimensional tape TM that states if a square with side *n* is symmetric with the main diagonal (please refer to Figure 3-8).

If useful, you can assume that the Machine has auxiliary memory, besides the bi-dimensional tape (for example one or more mono-dimensional tapes).

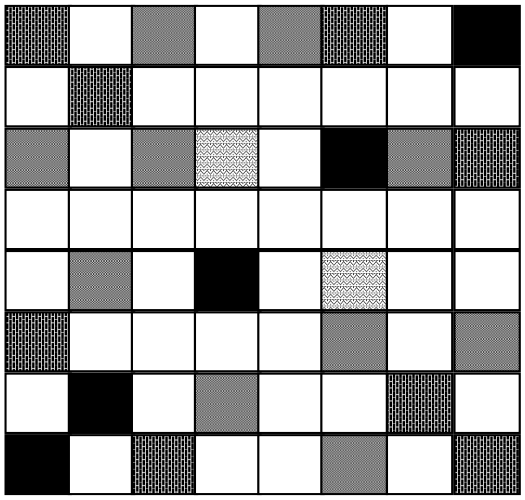


Figure 3-8 Bi-dimensional tape of a TM.

Please evaluate the asymptotic complexity (meaning related to the Θ) of the TM.

Solution

One of the feasible TM that solves the problem could have a service tape and be organized as follows:

1. The head of the bi-dimensional tape scans the first row and the last column:

read characters are copied onto the service tape and deleted from the bi-dimensional tape.

2. When reached the last cell at the right-bottom, it scans the last row (from right to left) and the first column (from bottom to top). For every read character verify that the content of the tape corresponds to it (the tape is used as a stack). The characters of both the service tape and the already scanned portion of the service tape are deleted.
3. When reached the left-top move to the cell on the right. If it is blank terminate with success, conversely start again from point a.

Every cycle takes a Θ(i), with i that goes from *n* to 1. So the whole takes $\Theta(n^2)$ elementary steps.

NB: if you were asked not to delete the content of the bi-dimensional tape, it would have been possible to substitute each read character with another one (with bijective correspondence) not originally present on the bi-dimensional tape, then scan again the whole tape in order to restore the initial situation. Given that this operation has $\Theta(n^2)$ complexity, the complexity of the whole program would not be modified.

Exercise 3.1.26

Given a set of alphabetic characters, in which each one could appear many times, provide an algorithm that produces, as output, the same sequence ordered in an alphabetical order. Please state, justifying the answers, if the execution of the algorithm using a k-tapes Turing Machine is more convenient then the execution on a RAM (using the logarithmic cost criterion).

It is better if the proposed algorithm has the minimum complexity on both the above machines (ignoring the relation Θ).

Solution

The most simple and efficient algorithm counts the occurrences of every single character of the input tape; at the end of the reading, it writes an ‘a’ for every ‘a’ read, then writes every ‘b’ for every ‘b’ read and so on for all other alphabetic characters.

Using a Turing Machine with 26 tapes to implement this algorithm (we decide to use a 26-character alphabet), it is possible to use each tape as unary counter of a given character of the alphabet. Thus it is possible to obtain a complexity equal to Θ(n) that it is impossible to obtain with a RAM that has to “spend”, in the worst case, log(i) steps for the read of the i-th character and the increment of its relative counter.

Exercise 3.1.27

Consider the following languages' classes tell if for which one the recognition can be done by a RAM machine in the same time (ignoring the Θ relation) taking the constant cost criterion and the logarithmic cost criterion:

1. The regular languages (recognizable by a finite state automaton)
2. The context-free (recognizable by a pushdown automaton)
3. The Recursively enumerable languages (recognizable by a Turing Machine)

Solution

Only the regular languages: indeed, since they are recognizable by a finite memory, RAM used for their recognition does not need an amount of data-dependent memory (data that in turn are representable by a sequence of elements belonging to a finite set). Therefore any operation, even by a logarithmic cost criterion, takes a limited amount of time a priori .

Instead some not-regular languages take a recognition time that, evaluated by logarithmic cost criterion, is necessarily greater than of the complexity evaluated by constant cost criterion.

For instance the language $L = \{wcw^R | w \in \{a,b\}^*\}$ requires storing the entire string w to be able to compare it with its reflected. The storing of i-th character of w takes, by a logarithmic cost criterion, a proportional time to at least log(i). Notice that is possible to store also the entire string w in a single cell that contains an integer (each bit of the integer represents a character of w) but this would require, for each character read, an operation whose time of execution for RAM is proportional to the length of the cell's content and then to the logarithm of its value.

Therefore the recognition of L by the RAM taking the logarithmic cost criterion cannot be less than $\Theta(n \cdot \log(n))$, while a constant cost criterion is $\Theta(n)$.

Exercise 3.1.28

It is known that it is correct to talk about "Class of solvable problems (deterministically) in polynomial time" without specifying what type of machine will solve them (provided that we take into account a "reasonable cost criteria"). Can we talk about "Class of solvable problems (deterministically) in exponential time" (with a function of time complexity $f(n) \leq k^n$, where k is a constant) without specifying what type of machine will solve them?

Solution

The resolution of a problem by an abstract machine M_1 in a time $\leq f_1(n)$, if it is executed by an abstract machine M_2, lead a complexity $f_2(n)$ increased by the function $P(f_1(n))$, where P is an appropriate polynomial (Polynomial correlation theorem). So, if $f_1(n) \leq k^n$, $f_2(n) \leq P(k^n) \leq (k^n)^h = k^{h \cdot n} = (k^h)^n$, f_2 is still an exponential function.

Exercise 3.1.29

Briefly describe an abstract machine that recognize the language $L = \{(a^n b^n)^m | n \geq 1$ e $m \geq 1\}$. Evaluate its complexity.

Solution

The given language is generated by a general grammar (see the Exercise 1.7.12). The recognizer machine must have the power of a Turing Machine.

A Turing Machine able to recognize the language L could work as follows:

5. Read the first *n* 'a', writing *n* (in a unary format) on the temporary tape.
6. Read the next 'b' scanning the tape (from right to left) and verifying the occurrence.
7. Read the next 'a' scanning the tape (from left to right) and verifying the occurrence. Than go back to point 2.

It is easy to verify that this machine executes a constant number of operations for each read character and the time spent to do this is proportional to the length of the input string.

Exercise 3.1.30

Consider two sequences of words ordered in alphabetical order. It is possible to assume, for the sake of simplicity, that the words have limited length (for instance, 20 characters).

Evaluate the complexity of executing a *merge* of the two sequences in a unique ordered sequence using:

3. A Turing Machine
4. A RAM (using the Logarithmic Cost Criterion)

Details of the two machines are not required. Provide a sufficiently clear and precise description able to provide the required information for an equally clear and precise complexity analysis.

Solution

Both the machines must store (completely, in the worst case) the input sequences in order to compare them. Such operation requires a time complexity equal to $\Theta(n)$ for a Turing Machine, and $\Theta(n \cdot \log(n))$ for a RAM (using a logarithmic cost criterion). As usual, when the RAM stores a sequence of n elements in consecutive cells, it spends $\Theta(\log(i))$ to store the i-th element and $\Theta(n \cdot \log(n))$ to store the entire sequence.

Subsequent comparison could be perform in linear time by a Turing Machine, but necessarily requires Θ(n log(n)) for a RAM that is, in this problem, intrinsically worse with respect to the Turing Machine.

Exercise 3.1.31

Consider the function f(n) = (n!)!, in other words f(n) = factorial of the factorial of n. Consider now the following algorithm for the computation of f(n):

```
DOUBLEFACTORIAL(N):
      TEMP:=1;
      FOR I:= 1 TO N DO
            TEMP:=I*TEMP;
      RESULT:=1;
      FOR J:=1 TO TEMP DO
            RESULT:=J*RESULT
```

1. Evaluate, considering the dependency on the input value n, the time complexity of the execution of the DoubleFactorial algorithm using the constant cost criterion on a RAM (the coding of the algorithm in the RAM language it is not required).
2. Evaluate the spatial complexity of the DoubleFactorial algorithm using constant cost criterion on a RAM.
3. Evaluate the spatial complexity of the DoubleFactorial algorithm using logarithmic cost criterion on a RAM.
4. Is the time complexity, evaluated with the logarithmic cost criterion, Θ> with respect to the time complexity evaluated with the constant cost criterion multiplied by the logarithmic of n?

NOTE: all the questions must be intended as considering the order of magnitude of the complexity of the various functions identified by the relation Θ.

Solution

1. The time complexity of the DoubleFactorial algorithm executed on a RAM is defined by the second cycle, executed n! times. Using the constant cost criterion, every execution of the cycle has a constant cost; the time complexity is Θ(n!).
2. Since a finite number of cells are used, the spatial complexity is constant, and equal to Θ(1).
3. The spatial complexity of the DoubleFactorial execution on RAM, using a logarithmic cost criterion, is increased by the outcome cell that, at the end of the execution, will store the value (n!)!. Its size will be Θ(log((n!)!)) = Θ(n!˙log(n!)) = Θ(n!˙n˙log(n)), because log(m!) = Θ(m˙log(m)).
4. Since the time complexity, using the same cost criterion, is ≥ than the spatial complexity, the time complexity, evaluated with a logarithmic cost criterion is Θ> than the time complexity, evaluated with the constant cost criterion and multiplied by the logarithm of n.

Exercise 3.1.32

Consider the problem of recognizing the language L = $\{a^n b^{in} \mid n > 0, n \geq i > 0\}$ by a Turing Machine. Define a Turing Machine, without necessarily means of all the details, that recognizes L trying to minimizing the *spatial complexity*. Evaluate the obtained spatial and time complexity.

Solution

A three-tape Turing Machine that recognizes L can operate in the following way: M reads 'a' *n* times and stores *n* in binary on tapes 1 and 2. This takes a Θ(log(n)) space. The tape 1 contains the value *n*, while the tape 2 is used to count the possible values of i.

Iteratively:

1. Read ‘b’ *n* times. To do this, copy *n* from the tape 1 and 3 and, for each ‘b’ read, the value

   contained in the tape 3 is decremented. Once the n-th ‘b’ is read, the content of the tape 2 is

   decremented.
2. Accept, if no more ‘b’ are present on it.
3. If other ‘b’ are present and the tape 2 contains the value 0, the execution is stopped in a non-

   acceptance state.
4. Otherwise, jump to point 1

It is easy to check that each tape contains, at least, the value of n that has a spatial complexity of Θ(log(n)) (because it is coded in binary). Since the length of the input string is m = n + i*n and at most (in the worst case when i=n) is Θ (n^2), and at least (when i=1) is Θ (n), Θ (log(n)) = Θ (log(m)). The spatial complexity of M is logarithmic with respect to the input length.

The time complexity is defined by the execution of the point 1, repeated *i* times. The complexity of the point 1 is dominated by the decreasing of the value of the tape 3 that, in the worst case, is equal to *n*. Such complexity is Θ(log(n)). Thus, the complexity of the entire point 1 is Θ(n·log(n)) and the complexity of the whole algorithm is Θ(i·n·log(n)) = Θ (m·log(m)).

Exercise 3.1.33

Please state, justifying the answer, which of the following statements are true:

1. There are problems for which time complexity of the solution obtained by any Turing Machine is lower than any other solution obtained by a RAM machine.
2. There are problems for which time complexity of the solution obtained by any RAM Machine is lower than any other solution obtained by a Turing Machine.

3. There are problems for which time complexity of the solution obtained by an appropriate Turing Machine is lower than any other solution obtained by a RAM machine.
4. There are problems for which time complexity of the solution obtained by an appropriate RAM machine is lower than any other solution obtained by a Turing Machine.

NB. The complexity of a RAM machine should be evaluated by a logarithmic cost criterion

Solution

The first two statements are false: indeed, you can always "worsen" the performance of any machine: given a machine (of a certain category) that solves a certain problem, you can always construct a Turing Machine (of another category) with worse performance.

The other two statements are true:

- A Turing Machine (in linear time) can re-write, in reverse order a sequence of characters. This requires the storage of *all data* before writing; hence, the storage of the i-th char needs a time complexity equals to $\Theta(\log(i))$ in a RAM machine. The total time complexity will be, at least, $\Theta(n \cdot \log(n))$
- We know that a RAM machine needs a time complexity equals to $\Theta(\log(n))$ to read an element in the middle of a sequence of n characters; a Turing Machine needs, at least, $\Theta(n)$.

Exercise 3.1.34

3. Explain with which temporal complexity (according to relation Θ) it is possible to simulate a Turing Machine with K+1 heads. which has a complexity of T(n), using a machine with K heads.
4. Make a generalization using a machine with K-H heads with H<K which simulate one with K heads.

Solution part 1

In the worst case the distance between the K+1 heads and the other K heads during the working process of the machine with K+1 heads is about T(n). According to what asserted before, in order to access to the information related there would be necessary $\Theta(T(n))$ moves. The overall complexity is thus $\Theta(T^2(n))$.

Solution part 2

The complexity increment doesn't change reducing more on the number of heads.

In fact, in time T(n) it is possible to execute only one complete scan of the whole tape and then simulate the behaviors of the other ones, using only one head. So $\Theta(T^2(n))$ is enough in order to simulate a machine with K head using a machine with only one head. Thus, it is a fortiori using a machine with K-H head with H<K.

Exercise 3.1.35

Exploit which are the best complexity (according to relation Θ) obtainable executing the f(n) = 2˙n function calculus, using respectively:

- a Turing Machine with k tapes
- a RAM (with logarithmic cost criterion).

Solution

Considering to have a binary encoding of the number n, the RAM machine could execute the only one operation necessary to the execution of the function with the time of Θ(log(n)). The Turing Machine with k tapes could obtain the result copying the binary encoding of number n followed by a zero. The time cost is equal: Θ(log(n)).

In both cases, it is not possible to obtain something better because the RAM machine has to execute at least one operation, which depends on the input data, and the Turing Machine has to scan the input string in order to copy it.

Exercise 3.1.36

Consider the language $L = \{a^{n^2} \mid n \geq 1\}$ that must be recognized using a RAM machine. Evaluate its complexity without coding the RAM recognizer program using a logarithmic cost criterion.

Solution

Let us call m the length of the input string (it belongs to the language of the strings a^m with $m=n^2$).

The easiest way in which the recognizer machine could operate is the following:

1. Reads the input string and stores its length. This operation has a complexity equal to Θ(m˙log(m)) because the reading of each character must correspond to the update of a counter, or the creation of a new cell: in both cases, it is needed a time of log(m) for each read character.
2. Initializes the variable i to 1 and then, for increasing values of I, computes its squared value and verifies if such value is equal to m. This operation is repeated until i^2 exceeds m that is, at least, n times. The complexity of this phase is Θ(n˙log(n)).

The read phase is dominant, and the complexity of the whole program is Θ(m˙log(m)).

Exercise 3.1.37

Consider the problem of the recognition of the language L $= \{w \mid w \in \{a,b\}^*, \#(w,a) = \#(w,b)\}$, which means that the language is made of the string of the alphabet {a,b} with an equal number of "a" and "b". Define a Turing Machine and a RAM machine which are able to recognize the language described above. Evaluate the complexity (with the notation Θ) for the RAM machines using the logarithmic constant cost criterion.

Solution

A Touring Machine which is able to recognize L could be built simulating a stack automaton: "a" characters exceeding the number of "b" characters, in the string scanned until a certain point, are "stacked" (vice versa for the "b" respectively to the "a"). The reading of another "a" character will cause the addition of an extra element on the stack; vice versa another "b" character will cause the deleting of an element on the stack. The string is considered valid if at the end of the scan process the stack automaton contains only the starting symbol Z_0.

Temporal complexity of machine like the one described above, is equal to $\Theta(n)$.

An easiest way to solve the problem is to count (with unary encoding) the "a" characters on a tape and "b" characters on another tape and thus making a comparison between both the tapes at the end of the reading process. The temporal complexity is equal to $\Theta(n)$.

A RAM machine could work simulating the behavior of a Touring Machine, like the one described above. Otherwise it could use a counter of the number of the exceeding characters (one cell for "a" and one for "b") and update it in a similar way to stack automaton. In both cases the *temporal* complexity would be equal to $\Theta(n \cdot \log(n))$. In the first case in fact the length of the stack should be proportional to n and thus any access to the top element would cost, in the worst case, $\Theta(\log(n))$; in the second case every update of the counter would cost, in the worst case, $\Theta(\log(n))$ too, because the value of the counter could be proportional to n.

In the first case we have a space complexity equal to $\Theta(n)$ and a time complexity equal to $\Theta(\log(n))$.

Exercise 3.1.38

The following sentence is true or false? Justify briefly the answer.

"If it is possible to simulate the computation of a calculation model M1 with a computation model M2 ensuring a relation from the two time complexities equal to $T_{M2}(n) \le f(T_{M1}(n))$, then it is possible to simulate the execution of the computation model M2 with M1 ensuring the relation between the two time complexity $T_{M1}(n) \le f^{-1}(T_{M2}(n))$".

Solution

The sentence is false. In fact, it is possible to simulate a RAM using a Turing Machine with a relation $T_M(n) \leq T^2_R(n)$, but for sure it is not possible to simulate a Turing Machine using a RAM machine with the relation $T_R(n) \leq \sqrt{}T_M(n)$. Indeed, sometimes it could happen that $T_R(n) = T_M(n).\log(T_M(n))$.

Exercise 3.1.39

Please state, justifying the answer, if the following statements are true or false:

1. If it is possible recognize a recursive language with time complexity equals to $\Theta(f(n))$, then it is possible recognize its complement with the same time complexity.
2. If it is possible recognize a recursive enumerable language with time complexity equals to $\Theta(f(n))$, then it is possible recognize its complement with the same time complexity.

Solution

1. True: a Turing Machine that recognizes a recursive language L can be always converted in a Turing Machine that recognizes L^ modifying the last movement of the Turing Machine from “acceptance” to “non acceptance”. It doesn’t change its complexity.
2. False: the complement of a recursive enumerable language should not be recursively enumerable; then, no Turing Machine accept it.

Exercise 3.1.40

L1 and L2 are two generic languages. L1 is recognizable with a Turing Machine with K tapes with time complexity $\Theta(f1)$ but not with less complexity. Respectively L2 is recognizable with time complexity $\Theta(f2)$ but not with less complexity. Say, justifying the answer, which of the following statements are true:

1. You can always recognize, with the Turing Machine with K tapes, $L1 \cap L2$ with complexity $\Theta(f1+f2)$
2. You can always recognize, with the Turing Machine with K tapes, $L1 \cap L2$ with complexity $\Theta(\text{Max}\{ f1, f2\})$
3. You can’t recognize, with the Turing Machine with K tapes, $L1 \cap L2$ with complexity $\Theta<\Theta(f1+f2)$
4. Some answers of the previous questions change if you consider the RAM with logarithmic cost criterion as a model for calculating, instead of referring to the Turing machine with K tapes.

5. Some answers of the questions 1-3 change if you consider the single tape Turing machine as a model for computation, instead of referring to the Turing machine with K tapes.

Solution

1. True: it is enough to build a MT that simulates first M1 and then M2 and that accepts only if both agree: the complexity is the sum of two complexities (less than Θ).
2. True: it can be shown either building a MT that simulates M1 and M2 "in parallel" or noting that Θ(f1+f2) is Θ-equivalent to Max{Θ(f1), Θ(f2)}.
3. False: the intersection between two languages can be the empty language, recognizable in constant time even if the two original languages are very complex.
4. False, as well as the 5: the arguments remain valid regardless of the applied calculation model adopted (with the exception of the simulation "in parallel" with machine single tape).

Exercise 3.1.41

Write, without using the multiplication instruction, a piece of SMALG program that, given three natural numbers, returns 1 if one of the three numbers is the product of the other two, 0 otherwise.

NOTE: for the sake of simplicity, specify a piece of program assuming that the data and the results are stored in memory cells without considering the input/output operations. Please use x, y and w as names for the data and z for the result.

Evaluate the algorithm complexity with the constant and logarithmic cost criterion.

Solution

One of the possible programs that fulfill the requirements is given here:

```
if (x>y and x>w) then
    m := x;
    n1 := min(y,w);
    n2 := max(y,w);
fi
if (y>x and y>w) {
    m := y;
    n1 := min(x,w);
    n2 := max(x,w);
fi
if (w>y and w>x) {
    m := w;
    n1 := min(x,y);
    n2 := max(x,y);
fi
{ at this point m > n2 > n1 }
i := 0;
r := 0;
while i < n1 do
```

```
    r := r+n2;
    i := i+1;
od
{ at this point r = n2 * n1 }
if (r = m)
    z := 1
  else
    z := 0
fi
```

The complexity is determined by the while loop, considering that the min and max functions could be coded in order to have a complexity equal to Θ(1) with the constant cost criterion, and Θ(log(n)) with the logarithmic cost criterion.

Considering the constant cost criterion, the complexity of the piece of programs is Θ(N), where N is the minimum number between x, y and w.

With the logarithmic cost criterion, considering that r can become a big number, that requires a memory size proportional to the logarithm of its value, the complexity is equal to Θ(N*log(M)), where N is the minimum value between x, y and w, and M is the median.

Exercise 3.1.42

Consider the language generated by the following grammar:

$S \rightarrow AS \mid A$

$A \rightarrow aAb\mid ab$

Please state, justifying the answer, what is the spatial and time complexity of a RAM that recognizes the language, using the logarithmic cost criterion.

Solution

The easiest and most efficient way to recognize the string of a language using a RAM uses a counter of the a – that is incremented for each 'a' read and it is decremented for each 'b'; Every time a new string starts $a^{nj}b^{nj}$, the counters are set to 0. Each read of "a" or "b" has a constant time costs and each update costs Θ(log(i)); accordingly, the time complexity is Θ(n*log(n)), and the spatial complexity is equal to Θ(log(n)).

Exercise 3.1.43

Discuss the complexity of the automaton in 1.2.2, both in terms of time and space.

Solution

Time complexity

In the case of language without ']' the automaton does always exactly n or n+1 transitions, given an input string of length n belonging to the language.

In the case of language with ']' transitions can be in higher number, because there are also those who, against the reading of a ']' are used to "forget" the round brackets previously opened. However, it is evident that in the worst case - constituted of strings like (((((((((a] - transitions number is 2n-2, and always a number $\leq 2n$, because a string belonging to the language can have a maximum of n-2 parenthesis. Therefore the time complexity is Θ (n).

Spatial complexity

The worst case is still represented by strings like (((((((((a]. In these cases, said n the length of the string in input, on the stack are loaded n-2 symbols. Therefore, the space complexity is Θ (n).

Exercise 3.1.44

Consider the following languages:

$L_1 = \{ a^n b^{n/2} c^{n/2} \mid n \geq 1\}$ NOTE: the integer division, for instance 3/2 = 1.

$L_2 = \{ a^n b^n \mid n \geq 1\} \cup \{ a^n c^n \mid n \geq 1\}$

$L_3 = \{ a^n b^n \mid 1 \leq n \leq 20 \}$

$L_4 = \{ a^n b^n b^n \mid n \geq 1\}$

$L_5 = \{ a^n b^{2n} a^n \mid n \geq 1\}$

Please state, justifying the answer, what is the spatial and time complexity to recognizes the languages defined above. It is enough to identify the order of magnitude of the complexity, indicated by the relation Θ.

Solution

The language L_4 could be recognized using a DPDA, L_2 using a NPDA, L_1 and L_5 using a Turing Machine, and L_3 is a regular language.

Therefore, it is easy to recognize every language with a linear spatial and time complexity (excluding L_3, which has a spatial complexity equal to $\Theta(1)$). It is possible to obtain a spatial complexity in the order of log(n) accepting a worsening in the time complexity with the same order: it is sufficient to store (in binary) the number of read characters instead of storing the whole strings.

Exercise 3.1.45

Say, justifying the answer, which of the following statements are true?

1. There are two families of automata A1 and A2 such that A1 recognizes a family of languages most closely with the family of languages recognized by A2, but A2 recognize some languages with time complexity lower than what can be achieved with A1, compared with the order of magnitude determined by the relation Θ.

2. T_{RC} is used to indicate the time complexity of a generic RAM valued with constant cost criteria and T_{RL} indicates the time complexity of the same RAM valued with logarithmic cost criterion. Then:
   - ← $\Theta(T_{RL})$ is always $\leq \Theta(T_{RC} \cdot \log(T_{RC}))$
   - ← $\Theta(T_{RL})$ is always $\geq \Theta(T_{RC} \cdot \log(T_{RC}))$

Solution to point a

The statement is true. E.g. A1 = Single tape Turing machines, A2 = PDA. It is known that the PDA does not recognize languages recognized by MT. Conversely, the PDA can recognize $L = \{wcw^R\}$ with a time $\Theta(n)$, while the MT single tape may take a time $\Theta(n^2)$.

Solution point b1

B1 is false. Consider the well-known calculation of the factorial. The machine performs n multiplications, then $\Theta(T_{RC}) = \Theta(n)$. However, in each iteration multiplications involve a term of size n^i, then $\Theta(T_{RL}) = \Theta(n^2 \log(n))$. For further details see e.g. Exercise 3.1.3.

Solution point b2

B2 is also false. To check this, just consider a program that operates on data of constant size (or limited by a constant).

Consider, for example, a program that reads a sequence of bits and writes on the output device each bit read before, stopping when it reads three consecutive zeros.

The complexity of this program is $\Theta(n)$, whether you evaluate it with the criterion of constant cost or with the logarithmic cost criteria.

Indeed, the cost of operations contains the terms l(M[x]) and l(x), that are equivalent to a constant because they are limited by constant values (and also small).

The same is valid for a program that simulates an FSA. The complexity of this program is $\Theta(n)$, where n is the length of input string. The complexity doesn't change with the logarithmic cost criterion. Indeed, the cost of operations contains the terms l(M[x]) and l(x), that are equivalent to constants because they are limited by l(K1) and l(K2), where K1 and K2 are the number of states and the cardinality of the input alphabet: these are still constant values.

Exercise 3.1.46

Describe the algorithm for the RAM machine that computes the following function:

$f: N \rightarrow N;\ f(x) = \text{if } x \text{ even then } \sqrt{x^x} \text{ else } x^3$

and evaluates the order of magnitude of the time complexity using both constant and logarithmic cost criterion, assuming, as parameter, *n* equal to the binary value of the string length of the input *x*.

NOTE 1

It is not required to create the algorithm with an optimal complexity, but the evaluation of the exercise is based also on the reached complexity.

NOTE 2

It is not required to code the algorithm in detail: it is enough to describe it at a reasonable level of detail in order to enable a correct and precise evaluation of the complexity (considering the relation Θ) using, for instance, a high level pseudo-language SMALG-like or similar.

Solution

An easy algorithm that computes the function f is given in the following:

```
if x is even
then
        m := x/2; ris := 1
        for i := 1 to m do ris := ris*x
else ris := x*x*x
```

Clearly, the else branch can be neglected for the evaluation of the complexity (in the worst case).
A constant cost criterion, the time complexity is Θ (x). If we indicate with *n* the length of *x*, it is $\Theta(2^n)$.
Using the logarithmic cost criterion, the complexity of a single operation ris := ris*x is $\Theta(\log(x^i))$, then $\Theta(i*\log(x))$. The complexity of the loop is $\Theta(m^2*\log(x)) = \Theta(x^2*\log(x)) = \Theta(n*(2^n)^2) = \Theta(n*(2^{2n}))$.

It is possible to obtain a better complexity (here analyzed with the logarithmic cost criterion) using the following, more sophisticated, algorithm (we restrict the analysis to the case in which x is even and > 1):

```
m := x/2; ris := 1; xi := x;
        while m > 0do
begin
if m mod 2 = 1 then ris := ris* xi;
xi := xi*xi; m :=m div 2
end;
```

A single execution of the loop costs, for the maximum values of ris, Θ,(x*log(x)) like the previous algorithm. In this case, however, the cycle is executed, at least, log(m) + 1 times.
The total complexity is equal to $\Theta(x*\log^2(x))$, then $\Theta(n^2*(2^n))$.

Exercise 3.1.47

Describe briefly -without necessarily encode it in detail- an algorithm for the RAM machine (or any other high-level language like Pascal) which receives as input 3 integers, x, y, k, encoded as sequences of decimal digits, and separated by appropriate special characters, and which writes to the output tape the ratio x/y which is also encoded by a sequence of decimal digits using the special character ',' to separate the integer part from the decimal part by truncating the decimal part after k digits. The k digits of the decimal part must be written even if they are 0.

For example, if x = 10, y = 3, k = 12. The tape input could be represented as follows:

1	0	#	3	#	1	2	#						

And the result in turn could be represented on the tape output as follows:

3	,	3	3	3	3	3	3	3	3	3	3	3	3

Then evaluate the time complexity of the algorithm both with the criterion of constant cost and the logarithmic one.

Note:

1. You may assume that the RAM is equipped with the operations MULT, DIV and MOD calculating the product, the quotient and the reminder of the division between two numbers, respectively. Moreover, for simplicity, we can assume that their cost with the logarithmic criterion is similar to the cost of additive operators.
2. It isn't required a particularly sophisticated algorithm: the traditional algorithm used for manual calculation of the division is very satisfactory, too.
3. It is stressed that the complexity should be computed according to the length of the input string.

Solution

```
char in[MAX] = {"1000#3#13#"};

unsigned int c2ui(char c) {
  return ((unsigned int) c)-((unsigned int) '0');
}

void main () {
   unsigned int n=0;
   unsigned int x=0, y=0, k=0;
   unsigned int i = 0;
   unsigned int z, c, r;
   while (in[i] != '#')
     x=x*10+c2ui(in[i++]);
   i++;
   while (in[i] != '#')
     y=y*10+c2ui(in[i++]);
   i++;
   while (in[i] != '#')
     k=k*10+c2ui(in[i++]);
   z = x/y;
   printf("%d", x/y);
   c=0;
   printf(".");
   r = x%y;
   }
   while (c<k) {
     printf("%d", (r*10)/y);
     r=(r*10)%y;
     c++;
   }
   printf("\n");
}
```

Analysis of complexity using constant cost criterion:

The first three loops make an operation for each character read from the input, so their contribution is $\Theta(n)$.

For the rest, it is necessary to assess the number of characters that compose x, y and k:

k can easily be composed by a number of characters proportional to n (average n/3). In this case, the last loop is executed at most $\Theta(10^n)$ times.

It can be concluded that the complexity of the program is $\Theta(10^n)$.

Analysis of complexity using logarithmic cost criterion:

Initial cycles build, at least, a number with an order of magnitude equal to $\Theta(10^n)$, and the resulting cost is $\Theta(n \log 10^n) = \Theta(n^2)$.

It is easy to verify that the cost is still determined by the while loop. It is needed to consider that the operation included in the loop involve a term *r* that becomes, at least, in the order of magnitude of 10*y, in other terms $\Theta(10^{n+1})$. The cost of each operation is equal to $\Theta(\log 10^{n+1}) = \Theta(n)$. The cost of the last loop, and the last algorithm is $\Theta(n10^n)$.

Exercise 3.1.48

For each of the following statements say, explaining briefly the reasons, if they are true or false.

1. Given a generic computable function f there exists always an algorithm A that computes it, coded in the language of RAM, which complexity, valued at logarithmic cost criterion, is <= of an appropriate polynomial applied to complexity valued at constant cost criteria.
2. For some computable function f does exists an algorithm A that computes it, coded in the language of RAM, such that there not exists an equivalent algorithm A1 whose complexity, estimated at logarithmic cost criterion is <= an appropriate polynomial applied to the complexity of A valued at constant cost criteria.
3. Given any computable function f does exists an algorithm A that computes it, coded in the language of RAM, such that there not exist an equivalent algorithm A1 whose complexity, estimated at logarithmic cost criterion is <= of an appropriate polynomial applied to the complexity of A evaluated at constant cost criteria.

Solution

1. True: it is enough that A is an algorithm that simulates the behaviour of a MT that computes f.
2. True: it is well known that the function 2^{2^n} is computable in linear time at constant cost criteria, but it has spatial complexity - and thus time complexity - at least exponential in the logarithmic criterion.
3. False: some functions (such as most of those calculated by finite automata) have linear complexity both at constant and logarithmic cost criterion. It is therefore not possible to find an algorithm that has complexity according to the constant cost criterion Θ <that according to the logarithmic cost criterion one.

Exercise 3.1.49

Consider the language L defined as follow:

$s \in L \leftrightarrow \exists$ y,m ($s=c^{m}\cdot y \wedge m>0 \wedge y \in L_Y$)

$s \in L_Y \leftrightarrow s=\varepsilon \vee \exists$ x,m ($s=a^{m}\cdot x \wedge m>0 \wedge x \in L_X$)

$s \in L_X \leftrightarrow s=\varepsilon \vee \exists$ y,m ($s=b^{m}\cdot y \wedge m>0 \wedge y \in L_Y$)

Please state (justifying your answer) what is the minimum power automaton which recognizes L and specify, justifying your answer, by which time complexity the presented automaton recognizes L.

Solution

L is nothing but the following set of strings: $\{x | x=c^{m}(a^{k}.y)^{k}$, $m>0$, $k=\{0,1\}$ and $y\in\{a,b\}^*\}$. Then L is recognizable by a finite automaton. Its time complexity is therefore Θ (n).

Exercise 3.1.50

Say, briefly justifying the answers, if the following statements are true or false.

- The recognition of a contextual-free deterministic language can be performed by a k-tape Turing Machine with time complexity Θ (n).
- The recognition of a contextual-free deterministic language can be performed by a single-tape Turing Machine with time complexity Θ (n).
- The recognition of a contextual-free deterministic language can be performed by a RAM with temporal complexity Θ (n).
- The recognition of a regular language can be performed by a single-tape Turing Machine with time complexity Θ (n).
- The recognition of a regular language can be performed by a RAM with time complexity Θ (n).
- No contextual-free language can be recognized by a k-tapes Turing Machine with ***spatial*** complexity $\Theta <$ of $\Theta(\log(n))$.

Solution

1. True: the k-tape Turing Machine can simulate the deterministic pushdown automaton without changing the complexity. It is known that every pushdown automaton has a linear complexity
2. False: it was proven that the language wcw^{R} needs a complexity that is, at least, equal to $\Theta(n^2)$, using a single-tape Turing Machine
3. False: the language wcw^{R} needs, at least, $\Theta(n*\log(n))$
4. True: the single-tape Turing Machine behaves like a finite state automaton without any change on its complexity (at least of the last move)
5. True: also the RAM can easily simulate the finite state automaton with a fixed

amount of memory so that every step has a constant cost using the logarithmic cost criterion.

6. False: a regular language can be recognized using a fixed amount of memory

Exercise 3.1.51

Provide an estimation of the increase – within Θ relation – of the relation between time complexity in the following computation models, in case one of them is used to simulate the other and give brief explain the reasons of the answers.

1. A Turing Machine (MT) with k tapes simulates MT with k+1 tapes.
2. A Turing Machine (MT) with single tape simulates a RAM with time complexity computed with logarithmic cost criterion.

Solution

1. It is known that MT with 1 tape can simulate MT with K tapes with complexity $T_1(n) \leq T_k^2(n)$ (with the usual demonstration: memory of the machine with k tapes can be represented in the unique tape and simulation of the single step of the machine costs at most one complete scanning of the whole memory, that have dimension $\leq T_k(n)$). The result holds even if the total number of tapes decreases of one unit.
2. First of all, we can notice that MT with single tape can simulate MT with k tapes in quadratic time according to the original machine. This can simulate a RAM in quadratic time with respect to it; so MT with single tape can simulate a RAM in a time limited by the fourth power of the RAM complexity, evaluated with logarithmic cost criterion. Nevertheless, we can apply the demonstration that we have done to compute the relation of the complexity between RAM and Mt with k-tapes and we can notice that using the single tape of MT it is enough to do more steps in the tape to simulate a step in RAM. This number of steps, that is finite, does not change the order of the complexity relation, so it is still quadratic.

Exercise 3.1.52

Given the following language:

$L = \{a^n b^n c^n \mid n \geq 1\}$

Please detail how a single-tape TM can recognize L and analyze its complexity in terms of the relation Θ.

NB: solutions with lower complexity will be preferred.

Solution

A simple solution would be scanning more times the tape: for every scan the machine counts one a, one b and one c, for example replacing them with an '*': if finally all a,b and c have been replaced by '*' at the same time, the string is accepted. In this way one makes n scans and therefore complexity is $\Theta(n^2)$.

A more sophisticated solution is given "dividing by 2" the number of characters to be counted for each scan. First we describe how this can be accomplished when n is a power of 2, and then we eliminate this hypothesis.

In the first scan the machine converts the string $a^nb^nc^n$ into the string $(*a)^{n/2}$ $(*b)^{n/2}$ $(*c)^{n/2}$

In the second scan the string is changed into $(***a)^{n/4}$ $(***b)^{n/4}$ $(***c)^{n/4}$ and so on: if, finally, all a, b and c disappear at the same time the string is accepted.

The number of scans is $\Theta(\log(n))$ so the overall complexity is $\Theta(n.\log(n))$.

We now consider the case in which n is not a power of 2.

In this case in every scan the machine must also remember, using states, if the division by 2 left a reminder or not. If the comparison between the three exponents doesn't result in 3 even or 3 odd the string is immediately rejected. Otherwise it is converted in the following way:

In the first scan the machine converts the string $a^nb^nc^n$ into the string

$(*a)^{n/2}$ [\$]$(*b)^{n/2}$ [\$] $(*c)^{n/2}$[\$]

Where [\$] indicates the fact that the new symbol \$ is present or not if n is odd or even, respectively. Please note that in order to decide whether or not the symbol has to be written the machine could need one step backwards.

In the second scan the string is converted into

$(***a)^{n/4}$ [\$\$][\$]$(***b)^{n/4}$ [\$\$][\$] $(***c)^{n/4}$[\$\$][\$]

using the same brackets' meaning.

In the k-th scan the string will be converted into

$(*^{2k-1}a)^{n/(2k)}$ $[\$^{2k-1}]$ $[\$^{2k-2}]$ …[\$]$(*^{2k-1}b)^{n/(2k)}$ $[\$^{2k-1}]$ …[\$]$(*^{2k-1}c)^{n/(2k)}$ $[\$^{2k-1}]$ ...[\$]

In order to register the rest of the division using \$ characters, for each scan it could be necessary to execute $\Theta(2^k) = \Theta(n)$ steps backwards. Every scan does not involve more than $\Theta(n)$ steps. The number of steps is still $\Theta(\log(n))$ so the overall complexity is $\Theta(n*\log(n))$.

Exercise 3.1.53

Please state, justifying the answer, if the following statement is true or false:

There exist languages recognizable by means of FSA in time Θ -< (with lower order of magnitude of the temporal complexity) of the time requested by a single-tape TM.

Solution

The statement is false. Indeed a single-tape TM can emulate a FSA by behaving, by means of its control unit, in the exact way as the automaton. At most a final move will be necessary in order to state that the input string is terminated. But this does not change the order of magnitude of the complexity.

Exercise 3.1.54

Describe, briefly but with sufficient detail, a simple algorithm for calculating the function $2^{(n!)}$.

Then assess the order of magnitude, according to the relation Θ, of the spatial and temporal complexity of the algorithm execution with a RAM, both at constant and logarithmic cost criterion, depending on the length of the input data, assuming that it is encoded in binary.

Note that the algorithm does not need to be necessarily optimal for the execution complexity, it's not required a detailed encoding of the algorithm and you can use MULT and DIV RAM-operations, with their values of cost, although it is well known that these values are really optimistic.

Solution

A simple algorithm is a first cycle that, read n, compute the value m = n!; after that a further cycle, in which the index is varied from 1 to m, computes 2^m by successive multiplications by 2.

Note that you could optimize this cycle by repeating the square of the local variable x, initialized to 2, but if m is not a power of 2, then it should be a final stage for further multiplication by 2 that in the worst case would be of the same order of magnitude of m (in a balanced tree, the number of leaves is proportional to the number of internal nodes).

At constant cost criterion the spatial complexity of the algorithm is Θ(1) because a finite number of cells is enough for its execution.

Time complexity, evaluated in order of the input data n, is Θ(n)+ Θ(m) = Θ(n!). Given that the length of the representation of n in binary, called ll, is Θ(log(n)), the time complexity expressed in terms of' ll is $\Theta(2^{ll}!)$.

Spatial complexity with logarithmic cost criterion is Θ of the logarithm of the result, ie $\Theta(\log(2^{n!}))$, ie Θ(n!) and $\Theta(2^{ll}!)$.

Time complexity is determined by m-times repetition of a cycle whose single execution costs in proportion to the spatial complexity, i.e. the contents of the cell that holds the final result. Hence it is $\Theta(m*\log(2^{n!}))$, ie $\Theta((n!)^2)$ and $\Theta((2^{ll}!)^2)$.

Exercise 3.1.55

Describe with sufficient details, without necessarily specifying every detail, in which way a Turing machine with k tapes can recognize the language L = {ww, w ∈ {a,b}*}, analyzing the spatial and temporal complexity (for less then the order of magnitude determined by Θ-equivalence). Solutions that minimize the spatial complexity are preferred.

Solution

Following the same way that allows to recognize the language $\{wcw^R, w \in \{a,b\}^*\}$, we can store in a storage tape the value n of the length of the input string, coding it in binary; then we divide it by 2. At this point we store it in another tape with the value of i initialized to 1, also encoded in binary. Now we can test the equality of the characters in position i and n/2 + i and we proceed by varying i up to n/2.
The spatial complexity is therefore $\Theta(\log(n))$. The temporal one is $\Theta(n^2.\log(n))$: after the initialization, n/2 comparisons are in fact necessary; each of those requires n/2 steps; each of those implies the update of the counter stored in binary with logarithmic length compared to n.

Exercise 3.1.56

Describe briefly, without going into detail, a MT single tape (not with k tapes, with k = 1!) that recognizes the language $a^{n1}ba^{n2}ba^{n3} \ldots a^{ns}$ where $n_i >= n_{i-1}$. Analyze its complexity and compare it with those achieved through a MT with k tapes and through a RAM, with logarithmic cost.

Solution

A simple MT single tape, which recognizes the given language, can operate as follows:
an initial scan replaces the first a of each series with a special symbol (such as an asterisk). Subsequent scans do the same thing with the second a and so on. If *a* is not founded at the i-th series (while it was founded in the (i-1)-th one), the string is rejected.
If at a certain scan the previous i series have been completely replaced by asterisks, we continue with the remaining s-i series, until we finish the input string that is accepted only if we reach the end successfully.
It's clear that such a process has complexity Θ (n^2).
A machine with 2 tapes may instead operate easily with complexity Θ (n): it is enough to store the first n1 a on the tape 1; during the scan of the second series, we check that it is not shorter than the string on tape 1 and simultaneously we copy the second series on tape 2; we work then on the third series in the same way but with reversed tapes and so on until the end.
Finally, a RAM can simply use two counters to verify during the scan. Each counter update has a cost Θ (log(i)), where i is the value of the counter. The total complexity is therefore Θ (n.log(n)).

Exercise 3.1.57

Identify the best complexities (according to the relation Θ) that are obtainable to compute the function f(n) = 2*n using:

- A k-tape Turing Machine
- A RAM (using logarithmic cost criterion)

Solution

We assume that the number *n* is encoded in a binary format. In this case, the RAM can execute the single operation needed to compute the function in a time equal to Θ (log(n)).

The k-tape Turing Machine could compute the result by copying the binary encoding of *n* followed by a ‘0’. In this way it also requires a time equal to Θ (log(n)). Clearly, in both cases, it is not possible to obtain a better result because the RAM must execute an operation that depends on the input data, and the Turing Machine must scan the entire string to copy it.

Exercise 3.1.58

Consider the problem of recognizing the language L = {w| w ∈ {a,b}*, #(w,a) = #(w,b)}, i.e. the language consisting of strings on the alphabet {a,b} containing an equal number of a and b. Indicate roughly a Turing Machine and a RAM machine running the recognition of that language, and assess the order of magnitude (using the notation Θ) of time complexity. The RAM must adopt the logarithmic cost criterion.

Solution

A MT recognizing L can be constructed by simulating a pushdown automaton: the a (respectively, the b) in excess than b (respectively, in excess than a) in the string marked up to a certain point are "stacked". Further reading of a (respectively, b) causes the addition of an element on the stack; reading of b (respectively a) causes the deletion of an item. The string is accepted if, at the end of scan, the stack contains only the starting symbol Z_0.

Time complexity of such machine is clearly Θ(n).

A RAM could operate by simulating the behaviour of the MT above. Or it could keep a counter of the number of characters in excess (one cell for a and one cell for b) and update it in a manner similar to the stack. In both cases, however, time complexity would be Θ (n*log(n)). Indeed, in the first case, the length of the stack would be proportional to n and so any access to its top item would cost, in the worst case, Θ (log(n)); in the second case any update of the counter would cost, in the worst case, Θ (log(n)) too, because the value of the counter itself could be proportional to n.

Note that in the first case there would be a spatial complexity Θ (n) while in the second case it would be Θ (log(n)).

Exercise 3.1.59

Please state, justifying the answer, which time complexity (according to the relation Θ) is needed to simulate a K+1-head Turing Machine that has a complexity equal to T(n), using a K-head machine.

Generalize the answer to the case in which a machine with K-H heads (with H<K) simulates another one with K heads.

Solution

In the worst case, the distance between the (K+1)-th head and the other K, during the activity period of the (K+1)-head machine, has an order of magnitude equal to T(n). Therefore, to access the information, $\Theta(T(n))$ steps are needed. We deduce that the total complexity is equal to $\Theta(T^2(n))$.

The increase of complexity does not change when the number of heads is decreased. Therefore, in a time equal to T(n), it is possible to execute a scan of the complete tape and, subsequently, to simulate the behaviour of every head using only one head. Hence, $\Theta(T^2(n))$ is enough to simulate the K-head machine using another machine with a single head, and then, obviously, this is possible also with a machine with K-H heads (with H<K).

Exercise 3.1.60

A grammar is *linear* if its productions are similar to:

$A \rightarrow xBy$ or $A \rightarrow x$, where $A, B \in V_N$, $x, y \in V_T^*$.

Please state, briefly justifying the answer, what is the order of magnitude of time complexity of a Turing Machine that, for a *predetermined* linear grammar, named G, establishes if, given two generic strings α e β, where $S \Rightarrow^* \alpha$, the relation $\alpha \Rightarrow. \beta$ exists.

NOTE: if G is linear, and $S \Rightarrow^* \alpha$, then α is a string like xBy – or $\alpha \in V_T^*$.

Please state what will be the complexity if the problem is solved using a RAM evaluating the complexity using the logarithmic cost criterion.

Solution

The Turing Machine can use the following algorithm: it reads α and stores it in a tape (if $\alpha \in V_T^*$.., $\alpha \Rightarrow. \beta$ if and only if $\alpha = \beta$). Therefore, if α is similar to xBy, reads β and compares it with α verifying the equality until B is found in α. At this point, it verifies that β contains the right part of some productions of G which have B as left part and the remaining parts of the two strings are equal. The entire algorithm can be executed in a time equal to $\Theta(n)$, where *n* is the length of the two strings.

The algorithm, executed with a RAM, would require a time equal to $\Theta(n.\log(n))$ due to the need of storing α in subsequent memory cells (the i-STORE requires a time proportional to log(i)).

4. Formal Semantics

Exercise 4.1.1

Provide the operational semantics of the **repeat-until** construct available in several programming languages such as Pascal, Modula-2 and Ada. These semantics can be provided by defining a proper set of syntactic rules that enrich the SMALG grammar and translation mechanisms in the SMALM language.

Solution

Let be <RepeatStat> ====>* **repeat** List **until** Condition

the (abbreviated) syntax that defines the construct.

It is translated into SMALM+ as follows

```
    L
    C
    JZ n1
```

where:

- L is the list of SMALM+ instructions that translates List
- C is the list of SMALM+ instructions that translates Condition (it is that the accumulator at its end contains 1 or 0 depending on whether Condition is true or false)
- n1 is the first instruction of L.

Exercise 4.1.2

Write a program that calculates the SMALG+ function f(n) = n! recursively. Provide the operational semantics with the translation into SMALM+. Describe the status of SMALM+ during the execution of calculus of the factorial of 2.

Solution

The recursive SMALG+program that computes n! is the following:

```
var n,fatt;
begin
   read n;
   call factor(n, fatt);
   write fatt
end
procedure factor(i, f);
   var j,k
```

```
begin
   if i = 0   then f := 1
              else j := i -1; call factor(j, k); f := i *k
   fi
end
```

The translation in SMALM+ of the program in SMALG+ is the following:

```
1.     PUSH     n
2.     PUSH     fatt
3.     READ
4.     STORE    n         {read n}
5.     PUSH     retadd    {call to the factorial function}
6.     LOAD     n
7.     PUSH     i
8.     STORE    i
9.     LOAD     fatt
10.    PUSH     f
11.    STORE    f         {end of transmission of input parameters}
12.    PUSH     j
13.    PUSH     k
14.    LOAD@    PC
15.    ADD=     4
16.    STORE    retadd
17.    JUMP     32        {32 is the first instruction of the code related to the procedure}
18.    POP
19.    LOAD
20.    POP
21.    STORE    fatt
22.    LOAD
23.    POP
24.    STORE    n         {end of transmission of output parameters}
25.    POP                {end of procedure call}
26.    LOAD     fatt
28.    WRITE              {write fatt}
29.    POP
30.    POP
31.    HALT               {end of the part related to the main program}
                          {declaration of the factor procedure}
32.    LOAD     i         {evaluation of i = 0}
33.    SUB=     0
34.    JZ       37
35.    LOAD=    0
36.    JUMP     38
37.    LOAD=    1
38.    JZ       42
39.    LOAD=    1         {then}
40.    STORE    f         {f :=1}
41.    JUMP     69
42.    LOAD     i         {else}
```

```
43.     SUB=     1
44.     STORE    j        {j := i -1}
45.     PUSH     retadd   {start of the recursive call}
46.     LOAD     j
47.     PUSH     i
48.     STORE    i
49.     LOAD     k
50.     PUSH     f
51.     STORE    f
52.     PUSH     j
53.     PUSH     k
54.     LOAD@    PC
55.     ADD=     4
56.     STORE    retadd
57.     JUMP     32
58.     POP      2
59.     LOAD
60.     POP
61.     STORE    k
62.     LOAD
63.     POP
64.     STORE    j
65.     POP
66.     LOAD     i
67.     MULT     k
68.     STORE    f        {f := i*k}
69.     LOAD     retadd   {retadd is copied into the accumulator}
70.     JUMP@    ACC      {return to the call point}
```

The essential points of the previous program execution, corresponding to the reading of the value 2 for n, are indicated by the sequence of figures 4.1.a, b, c, d below.

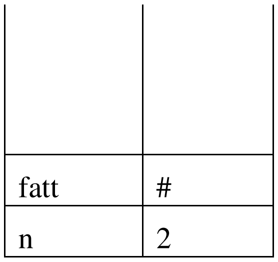
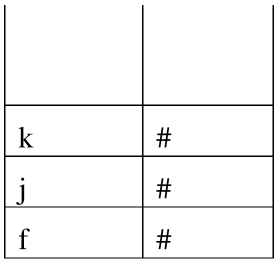


Figure 4.1.a Execution state just before the procedure call.



i	2
retadd	18
fatt	#
n	2

Figure 4.1.b State of execution at the beginning of execution of the first procedure call.

k	#
j	#
f	#
i	1
retadd	58
k	#
j	1
f	#
i	2
retadd	18
fatt	#
n	2

Figure 4.1.c State of execution at the beginning of execution of the second procedure call.

k	1
j	0
f	1
i	1
retadd	58
k	#
j	1
f	#
i	2
retadd	18
fatt	#
n	2

Figura 4.1.d State of execution at the end of execution of the second procedure call.

Exercise 4.1.3 (**)

Given $\{f_i\}$ as a chain of computable functions defined on a domain D^. The enumeration of $\{f_i\}$ is effective, which means that exist an algorithm able to build a Turing Machine M_i that computes f_i for each i.

Prove that the minimum upper limit of $\{f_i\}$, $f = LUB\{f_i\}$, is a computable function too.

Solution

For each $x \in \hat{D}$ $f(x) = \perp$ if and only if $f_i(x) = \perp$ for each i; $f(x) = y \neq \perp$ if and only if exists an i' that $f_{i'}(x) = y$ (and, in such case, $f_i(x) = y$ for each $i > i'$).

Build an algorithm A that, for a fixed x, and given a generic *i*, simulates the computation of a machine M_i for, exactly, i steps. *A* terminates for each x, and for each i. Execute *A*, given x, for increasing values of i. *A* terminates if and only if $f(x) \neq \perp$. In addition, if *A* terminates, the value that it computes is $f_{i''}(x)$ for each $i'' \geq i'$; then $f_{i''}(x) = f_{i'}(x) = f(x)$.

Exercise 4.1.4

a. Provide, through translation into the language SMALM, the operational semantics of the **loop** construct adopted by Modula-2 and other programming languages. Syntactically loop construct is the keyword **loop** followed by the loop body and the keyword **endloop**. Inside the loop you can find the keyword **exit** that determines the immediate exit from the cycle, resuming the execution from the first statement after the loop.

b. Say if SMALG* language, defined depriving SMALG of the **while** construct and equipping it in its place of the **loop** construct above, has the same power of Turing Machines.

c. Say if the halting problem for programs written in SMALG* is decidable.

Solution point a

Given S = **loop** SL **endloop** a generic loop of type **loop**, and L the sequence of SMALM instructions that provide the semantics of SL. Let n1 be the ordering number of the first instruction of L and n2 the ordering number of the first instruction that executes L in the program that translates the source program.

The semantics of S is given in the following:

```
n1:     L
        JUMP n1
n2:     ...
```

An eventual additional **exit** instruction inside SL is translated into

```
        JUMP n2
```

Solution point b

The traditional **while** cycle could be simulated by the **loop** cycle as follows:

```
while C do SL od
```

becomes

```
loop
      if C then exit fi
      SL
endloop
```

Solution point c

Given the equivalence with SMALG, thus also with the Turing Machine, the termination of the SMALG* programs is an undecidable problem.

5. Specification and analysis of programs and systems

5.1 Hoare's method for program properties verification

Exercise 5.1.1

Demonstrate the following theorem

```
{n>1}
begin i  := 1;
            z  := 1;
            while z ≤ n do
                  i  := i+1;
                  z  := i*i
            od
            sqrt  := i-1
end
```

$\{sqrt^2 \le n \wedge n < (sqrt+1)^2\}$

using the predicate I as loop invariant.

I: $z = i^2 \wedge (i-1)^2 \le n$

Solution

Given the composition rule IR1, the demonstration is divided into the following lemmas.

Lemma 1. {n>1} i := 1; z := 1 {I}.

Lemma 2. {I} **while ... od** $\{I \wedge z > n\}$.

Lemma 3. $\{I \wedge z > n\}$ sqrt := i-1 $\{sqrt^2 \le n \wedge n < (sqrt+1)^2\}$.

Proof of lemma 1.Applying two times the assignment axiom the result is the following:

$\{0 \le n\} \equiv \{1=1^2 \wedge (1-1)^2 \le n\}$ i:=1 $\{1=i^2 \wedge (i-1)^2 \le n\}$ z:=1 $\{z=i^2 \wedge (i-1)^2 \le n\}$

Lemma 1 is based upon the fact that $n>1 \rightarrow n \ge 0$ and on the composition rule of IR1

Proof of lemma 2.Based on the while rule (IR4) we need to demonstrate the following:

$\{I \wedge z \le n\}$ i := i+1; z := $i_* i$ {I}

Applying two times the assignment axiom we obtain:

$\{i^2 \le n\}$ i:= i+1 $\{i*i=i^2 \wedge (i-1)^2 \le n\}$ z:= i*i m$\{z=i^2 \wedge (i-1)^2 \le n\}$

.

Clearly, $I \wedge z \leq n \rightarrow i^2 \leq n$, which means $(i-1)^2 \leq n \wedge i^2 = z \wedge z \leq n \rightarrow i^2 \leq n$, thus lemma 2 with the implication rule.

Proof of lemma 3.Applying the assignment axiom we obtain:

$\{(i-1)^2 \leq n \wedge n < i^2\}$ sqrt:=i-1 $\{sqrt^2 \leq n \wedge n < (sqrt+1)^2\}$

and we can easily demonstrate $I \wedge z > n \rightarrow \{(i-1)^2 \leq n \wedge n < i^2\}$.

In fact $z = i^2 \wedge (i-1)^2 \leq n \wedge z > n \rightarrow (i-1)^2 \leq n \wedge n < z = i^2$. Lemma 3 is proved, by the implication rule.

Exercise 5.1.2

Write a program that evaluates the least common multiple of two positive integer x and y. Demonstrate the correctness of the program.

Solution

```
mcm:  begin
            z := 1;
            while z mod x ≠ 0  or  z mod y ≠ 0 do
                  z := z+1
            od
        end
```

We will prove the partial correctness of the program mcm in relation to the predicates

Pre: $x \geq 1 \wedge y \geq 1$ and

Post: $z \bmod x = 0 \wedge z \bmod y = 0 \wedge \forall w\, (1 \leq w < z \rightarrow w \bmod x \neq 0 \vee w \bmod y \neq 0)$.

We choose as loop invariant the predicate

I: $\forall w\, (1 \leq w < z \rightarrow w \bmod x \neq 0 \vee w \bmod y \neq 0)$

Partial correctness Theorem of mcm:

{Pre} mcm {Post}

According to composition rule IR1 and consequences rule IR2, the demonstration is based on the following three lemmas:

Lemma 1. {Pre} z := 1 {I}.

Lemma 2. {I} **while .... od** $\{I \wedge \neg c\}$, dove $c = z \bmod x \neq 0 \vee z \bmod y \neq 0$.

Lemma 3. $I \wedge \neg c \rightarrow$ Post.

Proof of lemma 1.For the assignment axiom (A1) we have that

{true} z := 1 { $\forall w\, (1 \leq w < z \rightarrow w \bmod x \neq 0 \vee w \bmod y \neq 0)$ }

Because, substituting 1 instead of z in I, the premise of the implication is false for any given value of w.

According to the logic theory we have that Pre $\rightarrow$ true, thus lemma 1 is derived from the consequences rule of (IR2).

Proof of lemma 2.According to the while rule (IR4) we just need to proof that:

{I ∧ c} z := z+1 {I}

For A1 we have that

$$I' \equiv \left\{ \forall w \left(1 \le w \le z \rightarrow \begin{pmatrix} w \bmod x \ne 0 \\ \vee \\ w \bmod y \ne 0 \end{pmatrix} \right) \right\} z := z+1 \left\{ \forall w \left(1 \le w < z \rightarrow \begin{pmatrix} w \bmod x \ne 0 \\ \vee \\ w \bmod y \ne 0 \end{pmatrix} \right) \right\} \equiv I \, .$$

We also have I ∧ c → I', then

$$\begin{pmatrix} \forall w \left(1 \le w < z \rightarrow \begin{pmatrix} w \bmod x \ne 0 \\ \vee \\ w \bmod y \ne 0 \end{pmatrix} \right) \\ \wedge \\ (z \bmod x \ne 0 \vee z \bmod y \ne 0) \\ \\ \\ \end{pmatrix} \rightarrow \forall w \left(1 \le w \le z \rightarrow \begin{pmatrix} w \bmod x \ne 0 \\ \vee \\ w \bmod y \ne 0 \end{pmatrix} \right) .$$

The lemma 2 is proved, according to the consequence rule IR2.

Proof of lemma 3. It's easy to proof the fact that I ∧ ¬c ≡ Post, hence, *a fortiori*, I∧¬c→Post, since

$$\neg \begin{pmatrix} z \bmod x \ne 0 \\ \vee \\ z \bmod y \ne 0 \end{pmatrix} \equiv \begin{pmatrix} z \bmod x = 0 \\ \wedge \\ z \bmod y = 0 \end{pmatrix} .$$

The stop of the program is proved as follows. We choose, a well ordered set the pair□N, >□of natural number with the relation '<' (minority), and as a function on the space program variable the following

f(x, y, z) = x·y - z.

Thus we have that the function is like *f*: $N^3 \rightarrow N$ (the codomain of *f* is N and not Z, the set of integer is a consequence on the invariant z ≤ x·y). The condition f(x', y', z') > f(x", y", z") is an obvious consequence of the fact that variable x and y remain unchanged while z is incremented.

Exercise 5.1.3

Define a SMALG program that verifies if a given value *x* is prime and demonstrate that it is correct according to the precondition *Pre*: *x*>1 and the post condition:

Post: $(pr = 0 \rightarrow \exists \psi(1 < \psi < \xi \wedge \xi\mu\ o\delta\ \psi = 0) \wedge (\pi\rho = 1 \rightarrow \forall \psi(1 < \psi < \xi \rightarrow \xi\mu\ o\delta\ \psi \ne 0)$

Solution

```
PRM:  begin
            i := 2;
            pr := 1;
            while i<x do
                  if x mod i = 0 then
                        pr := 0
                  fi
                  i := i+1
            od
      end
```

Theorem: PRM is partially correct: {Pre} PRM {Post}.

Demonstration: using the loop invariant method

$$I:\left(\begin{array}{c} i \le \xi \\ \wedge \\ (\pi\rho = 0 \to \exists\psi(1 < \psi < \iota \wedge \xi\,\mu o\delta\ \psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi < \iota \to \xi\,\mu o\delta\ \psi \ne 0) \end{array}\right)$$

using the composition rules it is possible to apply the following Lemmas:

Lemma 1. {Pre} i := 2; pr := 1 {I}.

Lemma 2. {I} **while ... od** {I ∧ i≥x}.

Lemma 3. I ∧ i≥x → Post.

Proof of lemma 1. Apply two times the assignment axioms.

$$\left\{\begin{array}{c} i \le \xi \\ \wedge \\ (\forall\psi(1 < \psi < \iota \to \xi\,\mu o\delta\ \psi \ne 0) \end{array}\right\} \pi\rho := 1 \left\{\begin{array}{c} \iota \le \xi \\ \wedge \\ (\pi\rho = 0 \to \exists\psi(1 < \psi < \iota \wedge \xi\,\mu o\delta\ \psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi < \iota \to \xi\,\mu o\delta\ \psi \ne 0) \end{array}\right\}$$

$$\{2 \le \xi\} \equiv \left\{\begin{array}{c} 2 \le \xi \\ \wedge \\ (\forall\psi(1 < \psi < 2 \to \xi\,\mu o\delta\ \psi \ne 0) \end{array}\right\} \iota := 2 \left\{\begin{array}{c} \iota \le \xi \\ \wedge \\ (\forall\psi(1 < \psi < \iota \to \xi\,\mu o\delta\ \psi \ne 0) \end{array}\right\}$$

The lemma 1 follows from the consequence rule and from the fact that Pre ≡ x>1 → x≥2.

Proof of lemma 2. Using the *while* rule it is enough to demonstrate that

{I ∧ i < x} **if ... fi**; i := i+1 {I}.

Apply the assignment axiom.

$$I_1 \equiv \left\{\begin{array}{c} \iota < \xi \\ \wedge \\ (\pi\rho = 0 \to \exists\psi(1 < \psi \leq \iota \wedge \xi\mu\,o\delta\,\psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi \leq \iota \to \xi\mu\,o\delta\,\psi \neq 0) \end{array}\right\} \iota := \iota + 1 \left\{\begin{array}{c} \iota \leq \xi \\ \wedge \\ (\pi\rho = 0 \to \exists\psi(1 < \psi < \iota \wedge \xi\mu\,o\delta\,\psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi < \iota \to \xi\mu\,o\delta\,\psi \neq 0) \end{array}\right\} \equiv I$$

I

According to composition rule, we need to demonstrate that

{I ∧ i < x} **if ... fi** {I1}.

We use the **if-then** rule to reduce this demonstration to the one that we use to prove the following two lemmas:

Lemma 2.1 {I ∧ i < x ∧ x mod i = 0} pr := 0 {I1}.

Lemma 2.2 I ∧ i < x ∧ x mod i ≠ 0 → I1.

Proof of Lemma 2.1. Apply the assignment axiom to the **then** branch of the conditional instruction.

$$I_2 \equiv \left\{\begin{array}{c} \iota < \xi \\ \wedge \\ (\exists\psi(1 < \psi \leq \iota \wedge \xi\mu\,o\delta\,\psi = 0) \end{array}\right\} \pi\rho := 0 \left\{\begin{array}{c} \iota < \xi \\ \wedge \\ (\pi\rho = 0 \to \exists\psi(1 < \psi \leq \iota \wedge \xi\mu\,o\delta\,\psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi \leq \iota \to \xi\mu\,o\delta\,\psi \neq 0) \end{array}\right\}$$

It is easy to recognize that I ∧ i < x ∧ x mod i = 0 → I2, and then

$$\left(\begin{array}{c} i \leq x \\ \wedge \\ \big(pr = 0 \to \exists y(1 < y < i \wedge x \bmod y = 0)\big) \\ \wedge \\ \big(pr = 1 \to \forall y(1 < y < i \to x \bmod y \neq 0)\big) \\ \wedge \\ i < x \\ \wedge \\ x \bmod i = 0 \end{array}\right) \to \left(\begin{array}{c} i < x \\ \wedge \\ \big(\exists y(1 < y \leq i \wedge x \bmod y = 0)\big) \end{array}\right)$$

In fact, when the premise of the implication is satisfied, the existential quantification of the consequence is verified when y=i. The Lemma 2.1 is demonstrated through the consequence rule.

Proof of lemma 2.2. The implication I ∧ i < x ∧ x mod i ≠ 0 → I1 is verified:

$$\begin{pmatrix} i \le \xi \\ \wedge \\ (\pi\rho = 0 \to \exists\psi(1 < \psi < \iota \wedge \xi\mu\,o\delta\,\psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi < \iota \to \xi\mu\,o\delta\,\psi \ne 0) \\ \wedge \\ \iota < \xi \\ \wedge \\ \xi\mu\,o\delta\,\iota \ne 0 \end{pmatrix} \to \begin{pmatrix} \iota < \xi \\ \wedge \\ (\pi\rho = 0 \to \exists\psi(1 < \psi \le \iota \wedge \xi\mu\,o\delta\,\psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi \le \iota \to \xi\mu\,o\delta\,\psi \ne 0) \end{pmatrix}$$

when the premise of the implication is true, if pr=0, the value of y that satisfies the existential quantification in the premise satisfies also the consequence; if pr=1, the last term of the premise ensures that the universal quantification is valid in the consequence.

The lemma 2 is demonstrated.

Proof of lemma 3. Given

$(I \wedge i \ge x) \equiv$

$$\begin{pmatrix} i \le x \\ \wedge \\ \big(pr = 0 \to \exists y(1 < y < i \wedge x \bmod y = 0\big) \\ \wedge \\ \big(pr = 1 \to \forall y(1 < y < i \to x \bmod y \ne 0\big) \\ \wedge \\ i \ge x \end{pmatrix} \to \begin{pmatrix} i = x \\ \wedge \\ \big(pr = 0 \to \exists y(1 < y < i \wedge x \bmod y = 0\big) \\ \wedge \\ \big(pr = 1 \to \forall y(1 < y < i \to x \bmod y \ne 0\big) \end{pmatrix}$$

$$\to \begin{pmatrix} i = x \\ \wedge \\ \big(pr = 0 \to \exists y(1 < y < x \wedge x \bmod y = 0\big) \\ \wedge \\ \big(pr = 1 \to \forall y(1 < y < x \to x \bmod y \ne 0\big) \end{pmatrix} \to$$

$$\begin{pmatrix} (pr = 0 \to \exists\psi(1 < \psi < \xi \wedge \xi\mu\,o\delta\,\psi = 0) \\ \wedge \\ (\pi\rho = 1 \to \forall\psi(1 < \psi < \xi \to \xi\mu\,o\delta\,\psi \ne 0) \end{pmatrix} \equiv \text{Post.}$$

Since the lemmas 1, 2 and 3 are proven, the theorem of partial correctness is also proved.

For the termination test, we choose a function that depends on the program variables like

f(x, i) = x - i. Since the invariant I ensures the condition i≤x, we have f(x, i) ≥0, and then $f: N^2 \to N$ and it is possible to take, as well ordered set, ⟨N, <⟩ and, obviously, f(i', x') > f(i", x"), because, in the body of the loop, *i* is incremented and *x* remains unchanged.

Exercise 5.1.4

Given an array of integer declared as follows

```
a: array [0..n-1] of integer;    (*n is a predefined
constant ≥0*)
```

where the elements a[k] are coefficient of *k*-order. A polynomial of *n-1*-order is:

$a[n-1]\ x^{n-1} + a[n-2]\ x^{n-2} + \ ....\ + a[1]\ x^{1} + a[0]$

The following program computes the value of the polynomial according to the value of the variable *x* using the well known Horner's rule:

```
HRN: begin
            s := 0;
            k := n;
            while k≠0 do
                  k := k-1;
                  s := s * x + a[k];
            od
     end.
```

Demonstrate the correctness of the program HRN with respect to the precondition Pre: n≥0 and the post condition Post: $s = \sum_{0 \le \iota < \nu} \alpha[\iota]\xi^{\iota}$

Solution

The following formula is used as loop invariant for the **while** loop:

I: $s \cdot \xi^{\kappa} = \sum_{\kappa \le \iota < \nu} \alpha[\iota]\xi^{\iota} \wedge \kappa \ge 0$

In the demonstration we use appropriate axioms that describe the proprieties of the summation operator "Σ", for instance $\sum_{i \in \varnothing} t_i = 0$.

The demonstration of the partial correctness theorem {Pre} HRN {Post} is divided, as usual, in three lemmas:

Lemma 1.	{Pre} s:=0; k:=n {I}.
Lemma 2	{I} **while ... do** {I $\wedge$ k=0}.
Lemma 3.	I $\wedge$ k=0 $\rightarrow$ Post.

Proof of lemma 1. Using the assignment axiom and the consequence rule, proving that Pre $\rightarrow I^{0,n}_{s,k}$ is enough. The implication is verified because $I^{0,n}_{s,k}$ is the formula 0·xn=0.

Proof of lemma 2. For the while rule it is enough to prove that

{I $\wedge$ k≠0} k := k-1; s := s * x + a[k] {I}.

Applying the assignment axiom we obtain

$$\left\{\begin{array}{c} k \geq 1 \\ \wedge \\ \sigma * \xi^{\kappa} + \alpha[\kappa - 1]\cdot \xi^{\kappa-1} \\ = \\ \sum\limits_{\kappa-1 \leq \iota < \nu} \alpha[\iota]\xi^{\iota} \end{array}\right\} \kappa := \kappa - 1 \left\{\begin{array}{c} \kappa \geq 0 \\ \wedge \\ (\sigma * \xi + \alpha[\kappa])\cdot \xi^{\kappa} = \sum\limits_{\kappa \leq \iota < \nu} \alpha[\iota]\xi^{\iota} \end{array}\right\} \sigma := \sigma * \xi + \alpha[\kappa] \left\{\begin{array}{c} \kappa \geq 0 \\ \wedge \\ \sigma \cdot \xi^{\kappa} = \sum\limits_{\kappa \leq \iota < \nu} \alpha[\iota]\xi^{\iota} \end{array}\right\}$$

and we have

$$I \wedge \kappa \neq 0 \equiv \sigma \cdot \xi^{\kappa} = \sum_{\kappa \leq \iota < \nu} \alpha[\iota]\xi^{\iota} \wedge \kappa \neq 0 \wedge \kappa \geq 0 \rightarrow \sigma * \xi^{\kappa} + \alpha[\kappa - 1]\cdot \xi^{\kappa-1} = \sum_{\kappa-1 \leq \iota < \nu} \alpha[\iota]\xi^{\iota} \wedge \kappa \geq 1$$

Because to both terms of the equality in the implication premise are easily applied the term $a[k-1]\cdot x^{k-1}$.

Proof of lemma 3. Expanding the implication I $\wedge$ k=0 $\rightarrow$ Post,

$$I \wedge \kappa = 0 \equiv \sigma \cdot \xi^{\kappa} = \sum_{\kappa \leq \iota < \nu} \alpha[\iota]\xi^{\iota} \wedge \kappa = 0 \rightarrow \sigma = \sum_{0 \leq \iota < \nu} \alpha[\iota]\xi^{\iota} \equiv \Pi o \sigma \tau$$

the equality in the implication consequent is obtained from the first equality of the premise by substitution of the value 0 with k.

The termination proof uses the function on the variables of the program f(k)=k, obtaining f:N$\rightarrow$N because I$\rightarrow$k≥0; considering a well ordered set, we take ⟨N, >⟩ and we have f(k') > f(k") because the variable *k* is decremented in the body of the loop.

Exercise 5.1.5

Prove the correctness of the following program. It computes, in the variable y, the m-th power (with m≥0) of the variable x:

```
     POT:  begin
                 y := 1;
                 z := x;
                 k := m;
                 while k≠0 do
                       if k mod 2 ≠ 0 then
                             k := k-1;
                             y := y *z
                       end;
                       k := k div 2;
                       z := z*z
                 od
           end.
```

Solution

In order to prove the partial correctness of the program, we define the input predicate Pre: m≥0 and the output predicate Post: $y=x^m$.The loop invariant is I: $y \cdot \zeta^{\kappa} = \xi^{\mu} \wedge \kappa \geq 0$. The demonstration is based on three lemmas.

Lemma 1. Pre $\to I_{y,z,k}^{1,x,m}$,it is easy to prove because $I_{y,z,k}^{1,x,m}$ ≡1·x^{m}=x^{m}∧m≥0 is equivalent to Pre.

Lemma 2. I ∧ k=0 → Post, it is easy to prove because I∧k=0 ≡ y·1=x^{m}∧0≥0 is equivalent to Post.

Lemma 3. {I} **while ... do** {I ∧ k=0}.Using the *while* rule, it is enough to prove {I ∧ k≠0} ...body of the while loop ... {I}.

We apply the assignment axiom to instructions after the **if** statement naming I1 the formula $I_{k}^{k\delta\iota v 2} \equiv \psi\, \zeta^{2(\kappa\delta\iota\varpi 2)} = \xi^{\mu} \wedge \kappa\delta\iota\varpi 2 \geq 0$, and to the instructions of the **if** statement naming I2 the formula $I1_{\kappa}^{\kappa-1} \equiv \psi\, \zeta \cdot \zeta^{2((\kappa-1)\,\delta\iota\varpi 2)} = \xi^{\mu} \wedge (\kappa-1)\,\delta\iota\varpi 2 \geq 0$.

According to the composition rule and the **if-then** rule, we need to prove that I ∧ k≠0 ∧ k mod 2 ≠ 0 → I2, and that I ∧ k≠0 ∧ k mod 2 = 0 → I1.

Expanding the first implication:

$$y \cdot \zeta^{\kappa} = \xi^{\mu} \wedge \kappa \geq 0 \wedge \kappa \neq 0 \wedge \kappa\mu\,o\delta 2 \neq 0 \to \psi\, \zeta \cdot \zeta^{2((\kappa-1)\delta\iota\varpi 2)} = \xi^{\mu} \wedge (\kappa-1)\,\delta\iota\varpi 2 \geq 0$$

assuming the following premise: if k mod 2 ≠ 0 then k-1 is even and 2((k-1) div 2)=k-1, and we can rewrite the implication as follows:

$$y \cdot \zeta^{\kappa} = \xi^{\mu} \wedge \kappa > 0 \wedge \kappa\mu\,o\delta 2 \neq 0 \to \psi\, \zeta^{\kappa} = \xi^{\mu} \wedge (\kappa-1)\,\delta\iota\varpi 2 \geq 0$$

that is obviously true.

Expanding the second implication

$$y \cdot \zeta^{\kappa} = \xi^{\mu} \wedge \kappa \geq 0 \wedge \kappa \neq 0 \wedge \kappa\mu\,o\delta 2 = 0 \to \psi\, \zeta^{2(\kappa\delta\iota\varpi 2)} = \xi^{\mu} \wedge \kappa\delta\iota\varpi 2 \geq 0 .$$

If the premise "k is even" is true and the formula can be reduced as:

$$y \cdot \zeta^{\kappa} = \xi^{\mu} \wedge \kappa > 0 \wedge \kappa\mu\,o\delta 2 = 0 \to \psi\, \zeta^{\kappa} = \xi^{\mu} \wedge \kappa\delta\iota\varpi 2 \geq 0$$

that is obviously true.

The lemma 3 and the theorem of partial correctness are proved.

For the termination proof we choose $f(\bar{x}) = \phi(\bar{\kappa}) = \kappa$; in this case, we have f:N→N because I → k≥0. In addition, $I_{\bar{x}}^{\bar{x}'} \wedge X_{\bar{\xi}}^{\bar{\xi}\ni} \to \phi(\bar{\xi}\ni)\ \phi\ \phi(\bar{\xi}\forall)$ because, if the variable *k* is divided by 2, eventually being decremented in a state in which *k*>0, the value of *k* decreases.

Exercise 5.1.6

Consider the program of the exercise 3.1.7.

1. Write a specification as pre and post-conditions and verify the partial correctness of the program among the specification
2. Verify the total correctness

Solution point 1

Set:

ord(a) = $\forall$h ($1 \le h \le n \rightarrow a[h+1] \ge a[h]$), the predicate that is verified if and only if the array *a* is ordered;

ver(h,k) = $1 \le h < k \le n \wedge a[h] + a[k] = d$, the predicate that is verified if and only if the elements in position h e k verify the condition required by the problem.

The partial correctness can be expressed by the following formula, where we consider a simplified version of the program:

```
{n≥2 ∧ ord(a)}
    i:=1; j:=n;
    while (a[i] + a[j] ≠ d ∧ (i<j)) do
    if a[i] + a[j] < d     then i:= i+1 else j:=j-1 fi

    od
{∃h,k ver(h,k) → ver(i,j)}
```

Note that the post condition is slightly stronger than necessary, because it also requires i<j and not only i $\neq$ j.

To find a loop invariant we observe that during the loop, if there are h e k which is ver(h,k), this pair must always be included in the part (i,j) of the array. Otherwise we can lose a solution and, in the case of unique solution, the program would fail. Furthermore, during the execution, i can never exceed j, and the array*a* remains ordered (there are no changes to its elements).

An invariant is: I = ord(a) $\wedge$ $1 \le i \le j \le n$ $\wedge \forall$h,k (ver(h,k) $\rightarrow$ $i \le h \wedge j \ge k$).

We show that at the exit of the loop remains the post condition:

(L1) I $\wedge\neg$(a[i] + a[j] $\neq$ d $\wedge$ i<j) $\rightarrow$ ($\exists$h,k ver(h,k) $\rightarrow$ ver(i,j)).

If$\neg\exists$h,k ver(h,k) then the post condition is verified and also (L1). If$\exists$h,k ver(h,k), the post condition is reduced to only ver(i,j). p e q are two values which is ver(p,q), so p<q. If I $\wedge\neg$(a[i] + a[j] $\neq$ d $\wedge$ i<j) is verified, then $i \le p \wedge j \ge q \wedge 1 \le i \le j \le n$ so $i \le p < q \le j$, ie i<j. Then a[i]+a[j] = d, because $\neg$(a[i] + a[j] $\neq$ d $\wedge$ i<j) and then is ver(i,j).

Now verify that I is an invariant, that:

(L2) {I$\wedge$ (a[i] + a[j] $\neq$ d $\wedge$ i<j)} **if** a[i] + a[j] < d **then** i:= i+1 **else** j:=j-1 **fi** {I}.

Apply the rule of **if**..**then**..**else**.

For the **then** branch, taking a backwards assignment by i:=i+1 we only need to show that:

$$ord(a) \wedge 1 \le i \le j \le n \wedge \forall h,k\big(ver(h,k) \rightarrow i \le h \wedge j \ge k\big) \wedge \big(a[i] + a[j] \neq d \wedge i < j\big) \wedge a[i] + a[j] \ge d$$
$$\rightarrow$$
$$ord(a) \wedge 1 \le i+1 \le j \le n \wedge \forall h,k\big(ver(h,k) \rightarrow i+1 \le h \wedge j \ge k\big)$$

Assuming that the antecedent is verified, being $1 \le i \le j \le n \wedge i<j \rightarrow 1 \le i+1 \le j \le n$ e a[i] + a[j] $\neq$ d $\wedge$ a[i] + a[j] < d $\rightarrow$ a[i] + a[j] < d, we only need to demonstrate:

(L2.1) $\forall$h,k (ver(h,k) $\rightarrow$ i+1≤h $\wedge$ j≥k).

If¬∃h,k ver(h,k), than the (L2.1) is verified. If∃h,k ver(h,k), being p e q a pair where ver(p,q). We suppose for absurd that the consequent of (L2.1) is falsefor h=p and k=q, hence i+1≤p $\wedge$ j≥k is false. Being i≤p and j≥q, the only possibility for this is that i=p. Hence a[i] +a[j] < d = a[p] + a[q] = a[i] + a[q], so a[j] < a[q]. This is absurd because the vector is ordered and j≥q.

For the **else** branch, we must demonstrate, proceeding similarly of the **then** branch:

$$ord(a) \wedge 1 \le i \le j \le n \wedge \forall h,k\big(ver(h,k) \rightarrow i \le h \wedge j \ge k\big) \wedge \big(a[i]+a[j] \ne d \wedge i < j\big) \wedge a[i]+a[j] < d$$
$$\rightarrow$$
$$ord(a) \wedge 1 \le i \le j-1 \le n \wedge \forall h,k\big(ver(h,k) \rightarrow i \le h \wedge j-1 \ge k\big)$$

We must demonstrate that, given two p and q where ver(p,q), is true that j-1≥q. If j-1<q, being j≥q, should be j=q and then a[i] + a[j] >d = a[p] + a[q] = a[p] + a[j]. Then it should be a[i] > a[p], that is absurd because the array is ordered and i≤p.

To terminate the demonstration we only need to show that {n≥2 $\wedge$ ord(a)} i:=1; j:=n {I}. Applying two times the rule of backwards substitution at I we obtain ord(a) $\wedge$ 1≤1≤n≤n $\wedge\forall$h,k (ver(h,k) $\rightarrow$ 1≤h $\wedge$ n≥k), that is equivalent at ord(a) $\wedge$ n≥1, that further implies at n≥2 $\wedge$ ord(a).

Solution point 2

For the termination proof we only use the loop. In this particular case it is sufficient to use an invariant J that is easier than the one used previously, because we do not need to prove program correctness but its termination. *I* still remain usable, but it makes the proof unreasonably harder. We define J = i≤j. It is clear that J is an invariant (this should be proved because I$\rightarrow$J does not guarantee that J is an invariant if also *I* is an invariant).

As well founded set W, we choose the set of the natural numbers and the relation >. In addition, we choose the a function f where, if $\overline{x}$ is a state vector, f($\overline{x}$) is j-i.

It is obvious that J $\rightarrow$ f($\overline{x}$) $\in$ W. We prove that if $\overline{x}$' is the state vector before the execution of the loop body, $\overline{x}$'' is the state after the previous execution and C is the condition of while loop, then $J_{\overline{x}}^{\overline{x}'} \wedge C_{\overline{x}}^{\overline{x}''} \rightarrow f(\overline{x}') > f(\overline{x}'')$.

Formally, it is necessary to prove the following theorem:

{i=i' $\wedge$ j=j' $\wedge$ i<j $\wedge$ a[i] + a[j] ≠ d}

if a[i] + a[j] < d **then** i:= i+1 **else** j:=j-1 **fi**

{i≤j $\wedge$ i'<j' $\wedge$ a[i'] + a[j'] ≠ d $\rightarrow$ j'-i'>j-i}

where the new values of i and j, after an execution of the loop body, the difference between them is decreased.

About the assignment axiom, applied to the **then** branch, it is enough to prove that

$$i = i' \wedge j = j' \wedge i < j \wedge a[i] + a[j] \neq d$$
$$\rightarrow$$
$$(i+1 \leq j \wedge i' < j' \wedge a[i'] + a[j'] \neq d \rightarrow j' - i' > j - i - 1)$$

Considering, as hypothesis, that $i = i' \wedge j = j' \wedge i < j \wedge a[i] + a[j] \neq d$ and then: $i+1 \leq j \wedge i' < j' \wedge a[i'] + a[j'] \neq d$, we have $j' - i' > j - i - 1$.

For the **else** branch, applying the assignment axiom to j:= j-1 we obtain the previous formula again.

It completes the proof of termination. Considering the point 1 of the exercise, the program is completely correct.

Exercise 5.1.7

Considering the following declarations:

```
type colore = (rs, bl);   (* rs represents the red
and bl the blue *)
var a: array [0..n] of colore;    (* n predefined
constant *)
```

where the array *a* contains a set of n+1 non-ordered and blue elements, the following program order the array dividing it in two adjacent regions of red (initial positions) and blue (other positions) elements.

```
SRT: begin
          m := 0;
          b := n+1;
          while m<b do
               if a[m]=rs
                    then  m := m+1
                    else  b := b-1;
                          a[m] := a[b];
                          a[b] := bl
               fi
          od
     end
```

Prove the correctness of the SRT program with respect to the input predicate

Pre: $n \geq 0 \wedge \forall i(0 \leq i \leq n \rightarrow a[i]=rs \vee a[i]=bl\}$

and the output predicate

Post: $\forall i(0 \leq i < b \rightarrow a[i]=rs) \wedge \forall i(b \leq i \leq n \rightarrow a[i]=bl\}$.

Solution

We can visualize the configuration of the array during the execution of the program as listed in Figure 5-1, where the array is divided in three portions: the leftmost is composed by red elements, the one in center is “mixed”, and the rightmost is entirely blue. The variables *m* and *b* demarcate the initial position of the mixed and blue zone, respectively.

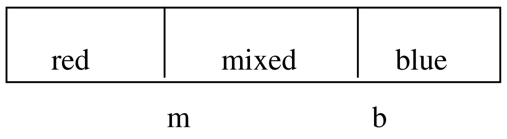


Figure 5-1 Division of the array during the execution of the program.

At the beginning, the mixed zone occupies the entire array (m=0 e b=n),while, at the end, the mixed zone will be deleted (m=b). We can write the loop invariant as follows:

$$
\text{I:}\quad
\begin{array}{l}
m \le \beta \wedge \\
\forall \iota(0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\
\forall \iota(\mu \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\
\forall \iota(\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda)
\end{array}
$$

We divide the proof of partial correctness in the following three lemmas:

Lemma 1. {Pre} m:=0; b:=n+1 {I}. We have:

$$
I_1 \equiv \left\{
\begin{array}{l}
0 \le \nu + 1 \wedge \\
\forall \iota(0 \le \iota < 0 \to \alpha[\iota] = \rho\sigma) \wedge \\
\forall \iota(0 \le \iota \le \nu \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\
\forall \iota(\nu < \iota \le \nu \to \alpha[\iota] = \beta\lambda)
\end{array}
\right\}
\mu := 0; \beta := \nu + 1
\left\{
\begin{array}{l}
\mu \le \beta \wedge \\
\forall \iota(0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\
\forall \iota(\mu \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\
\forall \iota(\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda)
\end{array}
\right\} \equiv I
$$

e I1 clearly implied by Pre.

Lemma 2. I $\wedge$ m≥b $\to$ Post. Expanding the formula we have:

$$
\left(
\begin{array}{l}
m \ge \beta \wedge \\
\mu \le \beta \wedge \\
\forall \iota(0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\
\forall \iota(\mu \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\
\forall \iota(\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda)
\end{array}
\right)
\to
\left(
\begin{array}{l}
\mu = \beta \wedge \\
\forall \iota(0 \le \iota < \beta \to \alpha[\iota] = \rho\sigma) \wedge \\
\forall \iota(\beta \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\
\forall \iota(\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda)
\end{array}
\right)
\to
$$

$$
\to
\left(
\begin{array}{l}
\mu = \beta \wedge \\
\forall \iota(0 \le \iota < \beta \to \alpha[\iota] = \rho\sigma) \wedge \\
\forall \iota(\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda)
\end{array}
\right) \equiv \Pi o\sigma\tau
$$

Lemma 3. {I} **while ... od** {I $\wedge$ m≥b}. Considering the while rule, it is enough to proof that

{I $\wedge$ m<b} **if ... fi** {I}

Considering the **if-then-else** rule, we introduce the lemmas 3.1 e 3.2.

Lemma 3.1. {I $\wedge$ m<b $\wedge$ a[m]=rs} m := m+1 {I}. We have

$$I_2 \equiv \{I \wedge \mu < \beta \wedge \alpha[\mu] = \rho\sigma\} \equiv \left(\begin{array}{l} \mu < \beta \wedge \\ \forall \iota (0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \alpha[\mu] = \rho\sigma \wedge \\ \forall \iota (\mu \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota (\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right)$$

furthermore

$$I_3 \equiv I_\mu^{\mu+1} \equiv \left(\begin{array}{l} \mu + 1 \le \beta \wedge \\ \forall \iota (0 \le \iota \le \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \forall \iota (\mu < \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota (\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right).$$

It is clear that I$_2$ implies I$_3$: the first term of I$_3$ is implied by the first of I$_2$; the second term of I$_3$ is implied by the second and the third of I$_2$; the third term of I$_3$ is implied by the fourth of I$_2$; the fourth of I$_3$ is equal to the fifth of I$_2$.

Lemma 3.2. {I $\wedge$ m<b $\wedge$ a[m] $\neq$rs} b := b-1; a[m] := a[b]; a[b] := bl {I}. Applying three times the assignment axiom we obtain

$$I_4 \equiv \left\{ \begin{array}{l} \mu < \beta \wedge \\ \forall \iota (0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \alpha[\beta - 1] = \rho\sigma \vee \alpha[\beta - 1] = \beta\lambda \wedge \\ \forall \iota (\mu < \iota < \beta - 1 \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota (\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right\} \beta := \beta - 1 \left\{ \begin{array}{l} \mu \le \beta \wedge \\ \forall \iota (0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \alpha[\beta] = \rho\sigma \vee \alpha[\beta] = \beta\lambda \wedge \\ \forall \iota (\mu < \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota (\beta < \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right\}$$

$$\left\{ \begin{array}{l} m \le \beta \wedge \\ \forall \iota (0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \alpha[\beta] = \rho\sigma \vee \alpha[\beta] = \beta\lambda \wedge \\ \forall \iota (\mu < \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota (\beta < \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right\} \alpha[\mu] := \alpha[\beta] \left\{ \begin{array}{l} \mu \le \beta \wedge \\ \forall \iota (0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \forall \iota (\mu \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \beta\lambda = \beta\lambda \wedge \\ \forall \iota (\beta < \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right\}$$

$$\left\{ \begin{array}{l} m \le \beta \wedge \\ \forall \iota (0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \forall \iota (\mu \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \beta\lambda = \beta\lambda \wedge \\ \forall \iota (\beta < \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right\} \alpha[\beta] := \beta\lambda \left\{ \begin{array}{l} \mu \le \beta \wedge \\ \forall \iota (0 \le \iota < \mu \to \alpha[\iota] = \rho\sigma) \wedge \\ \forall \iota (\mu \le \iota < \beta \to \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota (\beta \le \iota \le \nu \to \alpha[\iota] = \beta\lambda) \end{array} \right\} \equiv \{I\}$$

We immediately verifies that I $\wedge$ m<b $\wedge$ a[m] $\neq$rs $\rightarrow$ I4. In fact, in the implication

$$\left(\begin{array}{l} m \leq \beta \wedge \\ \forall \iota(0 \leq \iota < \mu \rightarrow \alpha[\iota] = \rho\sigma) \wedge \\ \forall \iota(\mu \leq \iota < \beta \rightarrow \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota(\beta \leq \iota \leq \nu \rightarrow \alpha[\iota] = \beta\lambda) \wedge \\ \mu < \beta \wedge \\ \alpha[\mu] \neq \rho\sigma \end{array}\right) \rightarrow \left(\begin{array}{l} \mu < \beta \wedge \\ \forall \iota(0 \leq \iota < \mu \rightarrow \alpha[\iota] = \rho\sigma) \wedge \\ \alpha[\beta - 1] = \rho\sigma \vee \alpha[\beta - 1] = \beta\lambda \wedge \\ \forall \iota(\mu < \iota < \beta - 1 \rightarrow \alpha[\iota] = \rho\sigma \vee \alpha[\iota] = \beta\lambda) \wedge \\ \forall \iota(\beta \leq \iota \leq \nu \rightarrow \alpha[\iota] = \beta\lambda) \end{array}\right)$$

the first term of the conclusion is equal to the fifth of the premise; the two second terms of the premise and conclusion are equal; third and fourth terms of the consequence are implied by the third term of the premise; the fifth term of the consequence is equal to the third of the premise.

The lemma 3.2, the lemma 3 and the theorem of partial correctness are proved.

Considering the termination proof, we use the following functions as variable of the program: f(b, m) = b – m; because I → m≤b we have $f{:}N^2 \rightarrow N$ and we can choose ⟨N, >⟩ as well ordered set. In that case we have f(b', m')>f(b", m") because, at each iteration of the loop *b* is decremented or *m* is incremented, and their difference increases.

Exercise 5.1.8

The following program performs a binary search on an array, which is supposed to be ordered.

```
....
var c: array [0..n-1] of integer;
....
begin
      l:=0; u:=n;
      while l<u do
            i := (l+u-1) div  2;
            if c[i] < x
                  then  l := i+1
                  else  u := i
            fi
      od
end
```

Please indicate the program's preconditions and post conditions and demonstrate its partial correctness and termination.

Solution

The precondition Pre: n>1 ∧∀i∀j(0≤i≤j<n → c[i] ≤c[j]) ≡ n>1 ∧ ord(c) states that the array has to be non-empty and ordered.

The post condition Post: ∀j (0≤j<l → c[j]<x) ∧ ∀i (l≤i<n → c[i]≥x) states (indirectly) that the searched element is, if present, in position *l*. The demonstration of partial correctness uses the loop invariant

I: l≤u ∧ ∀j((0≤j<l → c[j]<x) ∧ (u≤j<n → c[j]≥x)) ∧ ord(c)

and uses three lemmas.

Lemma 1. {Pre} l:=0; u:=n {I}. This because of the fact that $I_{l,u}^{0,n}$ ≡ 0 ≤ n ∧ ord(c), is obviously implied by the precondition.

Lemma 2. I ∧ l≥u → Post.

I ∧ l≥u ≡ l≤u ∧ ∀j((0≤j<l → c[j]<x) ∧ (u≤j<n → c[j]≥x)) ∧ ord(c) ∧ l≥u →

l=u ∧ ∀j((0≤j<l → c[j]<x) ∧ (u≤j<n → c[j]≥x)) ∧ ord(c) → Post

Lemma 3. {I ∧ l<u} **while ... od** {I}. We divide it in three more lemmas.

Lemma 3.1. {I ∧ l<u} i:=(l+u-1) div2 {I ∧ l<u ∧ i = (l+u-1) div2 }≡I1 is obviously valid.

Lemma 3.2. {I1 ∧ c[i]<x} l := i+1 {I}, so I1 ∧ c[i]<x →I_l^{i+1}, so

$$\begin{pmatrix} l \le \upsilon\wedge \\ \forall\varphi\begin{pmatrix}(0\le\varphi<\lambda\to\chi[\varphi]<\xi) \\ \wedge \\ (\upsilon\le\varphi<\nu\to\chi[\varphi]\ge\xi)\end{pmatrix}\wedge \\ \mathrm{op}\delta(\chi)\wedge \\ \iota=(\lambda+\upsilon-1)\,\delta\iota\varpi 2\wedge \\ \lambda<\upsilon\wedge \\ \chi[\iota]<\xi \end{pmatrix} \to \begin{pmatrix} \iota+1\le\upsilon\wedge \\ \forall\varphi\begin{pmatrix}(0\le\varphi<\iota+1\to\chi[\varphi]<\xi) \\ \wedge \\ (\upsilon\le\varphi<\nu\to\chi[\varphi]\ge\xi)\end{pmatrix}\wedge \\ \mathrm{op}\delta(\chi) \end{pmatrix}$$

from which

$$\begin{pmatrix} l<\upsilon\wedge \\ \forall\varphi(0\le\varphi<\lambda\to\chi[\varphi]<\xi))\wedge \\ \forall\varphi(\upsilon\le\varphi<\nu\to\chi[\varphi]\ge\xi))\wedge \\ \mathrm{op}\delta(\chi)\wedge \\ \iota=(\lambda+\upsilon-1)\,\delta\iota\varpi 2\wedge \\ \chi[(\lambda+\upsilon-1)\delta\iota\varpi 2]<\xi \end{pmatrix} \to \begin{pmatrix} (\lambda+\upsilon-1)\,\delta\iota\varpi 2+1\le\upsilon\wedge \\ \forall\varphi(0\le\varphi\le(\lambda+\upsilon-1)\,\delta\iota\varpi 2\to\chi[\varphi]<\xi))\wedge \\ \forall\varphi(\upsilon\le\varphi<\nu\to\chi[\varphi]\ge\xi))\wedge \\ \mathrm{op}\delta(\chi) \end{pmatrix}.$$

The implication consequence's first term is assured to be valid from the premise's first one; the implication consequence's second term is assured to be valid from the premise's fourth and sixth ones; the last two implication consequence's terms are already in the premise.

Lemma 3.3. {I1 ∧ c[i] ≥x} u:= i {I}, so I1 ∧ c[i] ≥x →I_u^i, so

$$\left(\begin{array}{l} l \leq \upsilon \wedge \\ \forall\varphi\left(\begin{array}{c} (0 \leq \varphi < \lambda \rightarrow \chi[\varphi] < \xi) \\ \wedge \\ (\upsilon \leq \varphi < \nu \rightarrow \chi[\varphi] \geq \xi) \end{array}\right) \wedge \\ o\rho\delta(\chi) \wedge \\ \iota = (\lambda + \upsilon - 1)\,\delta\iota\varpi 2 \wedge \\ \lambda < \upsilon \wedge \\ \chi[\iota] \geq \xi \end{array}\right) \rightarrow \left(\begin{array}{l} \lambda \leq \iota \wedge \\ \forall\varphi\left(\begin{array}{c} (0 \leq \varphi < \lambda \rightarrow [\varphi] < \xi) \\ \wedge \\ (\iota \leq \varphi < \nu \rightarrow \chi[\varphi] \geq \xi) \end{array}\right) \wedge \\ o\rho\delta(\chi) \end{array}\right)$$

from which

$$\left(\begin{array}{l} l < \upsilon \wedge \\ \forall\varphi\, 0 \leq \varphi < \lambda \rightarrow \chi[\varphi] < \xi) \wedge \\ \forall\varphi\, \upsilon \leq \varphi < \nu \rightarrow \chi[\varphi] \geq \xi) \wedge \\ o\rho\delta(\chi) \wedge \\ \iota = (\lambda + \upsilon - 1)\,\delta\iota\varpi 2 \wedge \\ \chi[(\lambda + \upsilon - 1)\,\delta\iota\varpi 2] \geq \xi \end{array}\right) \rightarrow \left(\begin{array}{l} \lambda \leq (\lambda + \upsilon - 1)\,\delta\iota\varpi 2 \wedge \\ \forall\varphi\, 0 \leq \varphi < \lambda \rightarrow [\varphi] < \xi) \wedge \\ \forall\varphi(\lambda + \upsilon - 1)\,\delta\iota\varpi 2 \leq \varphi < \nu \rightarrow \chi[\varphi] \geq \xi) \wedge \\ o\rho\delta(\chi) \end{array}\right)$$

in which the implication consequence's first term is assured to be valid from the premise's first one; the implication consequence's third term is assured to be valid from the premise's third and sixth one; the two terms are already in the premise.

This completes the partial correctness demonstration.

To demonstrate the termination, we choose as a function of the program's variables f(l, u) = u - l. We have that I → l≤u and so f:N^2→N, and we can take ⟨N, >⟩ as well ordered set. We have f(l', u')>f(l'', u'') because for each iteration of the loop the u-l decreases, either because the variable *l* increases or because the variable *u* decreases. Indeed if we take the **then** part of the conditional instruction, we have $l'' = \frac{\lambda\ni + \upsilon\ni + 1}{2} + 1$ and the condition *l''>l'*, is the same as $\frac{l' + \upsilon\ni + 1}{2} + 1 > \lambda\ni \equiv \lambda\ni + \upsilon\ni - 1 + 2 > 2 \cdot \lambda\ni \equiv \lambda\ni < \upsilon\ni + 1$ which is clearly implied by the loop's condition, *l'<u'*. Conversely if we take the **else part**, the condition *u''<u'* is the same as $\frac{l' + \upsilon\ni - 1}{2} < \upsilon\ni \equiv \lambda\ni + \upsilon\ni - 1 < 2 \cdot \upsilon \equiv \lambda\ni - 1 < \upsilon\ni$ which is clearly implied by the loop's condition, too.

Exercise 5.1.9

- Write the specific of a program which count the occurrences of character ‘a’ in an array A of characters of length n. Suggestion: first of all define, in a recursive manner, a function Occa(k) that count the occurrences number of char ‘a’ within first k elements of the array A. Then, use Occa(k) to built the specific of the program.

- Write a SMALG program that satisfies the specific of point 1.
- Prove the propriety of the program of point 2 with respect to the specific of point 1.

Solution point 1

Define Occa(k) in the following way:

Occa(0) = 0;

Occa(k) = **if** A[k] = ‘a’ **then** Occa(k-1) + 1 **else** Occa(k-1)

So the specific of the program is the following:

$\{n \geq 0\}$

P

{NumOcc = Occa(n)}

Where the variable NumOcc identify the result computed by P.

Solution points 2 and 3

P will be based upon a cycle that at each iteration will increase by 1 the value of NumOcc (inizialized to 0) if the test A[i] = ‘a’ will be successful; otherwise it will be hold steady.

Consequently, the proof of correctness of P will be based on an invariant of the following type:

I : {NumOcc = Occa(i-1)}

If i is the variable used for counting in the cycle, it is initialized to 1 and then incremented, according to the following schema:

```
i := 1;
while i ≤ n do
    ....
    i := i + 1
end;
```

The completion of the program and proof are leaved to the reader.

Exercise 5.1.10

Consider the following SMALG program:

```
begin
    z := x; exp := 1;
    while exp < n do z := z*z; exp := exp*2 od;
    while exp > n do z := z div x; exp := exp - 1 od
end
```

2. Prove, through the partial correctness proof, it calculates the function $z = x^n$, for positive values of x and n.
3. Prove the termination of the previous program.

4. Solution point a

Specific of the program is formalized by the following pair pre and post-condition:

$\{x > 0, n > 0\}$;

$\{z = x^n\}$

First cycle of the program, square repeatedly variable z producing values like $x^{exp}$with exp that is a power of 2. This is repeated until the exp reaches or exceed n. In the second case it is decreased by the second cycle through subsequent division of z by x until it become exactly n.

It is clear that the goal of the program is to keep the invariant $z = x^{exp}$ in order to reach the end of the program with exp=n. Moreover, at the end of the first cycle, exp≥ n. This suggest to split the proof of correctness as indicated by the following annotation of the program:

```
begin
{x > 0, n > 0}
     z := x; exp := 1;
     while exp < n do z := z*z; exp := exp*2 od;
     I1:   {z = x^exp and exp ≥ n}
     while exp > n do z := z div x; exp := exp - 1 od
{z = x^exp and exp = n}
end
```

I1 is clearly an invariant of the second cycle and, combined with the negation of exp > n, implies the post condition. Therefore, it remains to prove that at the end of the first cycle there is I1.

This is achieved by choosing as invariant of first cycle the assertion I2: $\{z = x^{exp}\}$.

Indeed I2 is certainly invariant for the first as for the second and, combined with the negation of the cycle condition (exp < n), implies (even coincides with) I1.

Finally, back propagation of I2 through the initial instructions produces the identity x = x.

Note that the proof of partial correctness does not highlight the fact that the first cycle assigns values to variable z that are always power of 2. Neither is used the precondition that is instead needed to ensure the termination (if n≤0 the program not ends).

Solution point b

Termination of first cycle of the program can be proved by noting that exp takes values that are multiples of 2; the condition of exit from the cycle is exp ≥n, so we can assume as well-ordered set W{2*i| 2*i ≤ 2*n}. Here we prove that if exp has value in W, at the beginning of the cycle, it is true also at the end of the cycle and exp assumes always increasing values (because initially has value >0). From the well ordering of W follows the termination.

Termination of second cycle is proved in an obvious way.

Exercise 5.1.11

Write a SMALG program that computes the function f(n) = n!. Prove the correctness of the program

Solution

The same program was used in the solution of the Exercise 3.1.3.

The loop invariant used to prove the correctness is

fatt = (i-1)! and $i \leq n+1$.

In fact, at the end of the loop we obtain $i = n+1$. Note that the test is successful if the precondition $n \geq 0$ is satisfied.

Exercise 5.1.12

Given the sequence E of n integers, the multiplicity of an integer x in E is the number of times x appears in E. A number x is a *majority* if its multiplicity is greater than $\lfloor n/2 \rfloor$. Create an algorithm that in linear time (with uniform cost criterion), given an E, either computes its majority or states that it does not exist. Please demonstrate the partial correctness of the method described.

Solution

The most obvious algorithm just counts the occurrences of all numbers in the sequence. The needed time is $\Theta(n^2)$. A better algorithm puts initially the numbers in order, and then counts the duplicates. The latter part can be computed in linear time, but the former has in general $\Theta(n \log n)$ complexity, except for particular cases.

An optimized linear time algorithm can be developed starting from the observation that, given a sequence *a* of n elements, if we have a majority and we eliminate two elements a_i e a_j, $a_i \neq a_j$, the majority does not change. Indeed, let us think that z is the majority, with a number of occurrences $k > \lfloor n/2 \rfloor$. If we delete a_i and a_j, the requested multiplicity for the majority becomes $\lfloor (n-2)/2 \rfloor$. At most one between a_i and a_j can be equal to z, so the new sequence, obtained deleting a_i e a_j, has a number of occurrences of z greater or equal than $k-1 \geq \lfloor n/2 \rfloor -1 \geq \lfloor (n-2)/2 \rfloor$: z is still the majority.

Then an algorithm can eliminate the couples of distinct element until it reaches a sequence that is small enough to state which is the majority (e.g., n=1 o n=2). But, *if the majority does not exist*, then the elimination of a couple of distinct elements can produce a sequence in which a majority exists. E.g., deleting 1 and 2 from the sequence without majority <1,5,5,2,2> we obtain the sequence <5,5,2> which has majority 5. But we can verify if a number is the majority for the initial sequence in linear time. Then the whole algorithm can compute first the elimination part, that ends with an index k of a candidate for the majority, followed by a verification part which controls whether the multiplicity of the x_k is greater than $\lfloor n/2 \rfloor$.

In order to simplify the algorithm and decrease the number of comparisons we can generalize the last observation stating that, given an i, if the portion of the sequence between 1 and i does not contain a majority, then the whole portion can be deleted from the sequence and the majority does not change. Indeed it is possible to eliminate all couples of elements in this portion, because if there is not a majority there always exists a way to match one element with another one of different value. The following algorithm uses this property to eliminate portions of the sequence. In order to be strict we do not calculate the number of occurrences of the majority candidate:

```
begin
      M:=1; i:=2; k:=1;
      while i<=n do
            if M=0
                  then k:=1; M:=1;
                  else if a[k] = a[i] then M:=M+1 else M:=M-1 fi
            fi
            i := i+1
      od
      m := -1;
      if M >0 then "conta numero occorrenze O di a[k]"
                  if O > n div 2 then m:= a[k] fi
      fi
      write m;
end
```

On each iteration of the first cycle variable M represents the difference between the count of "a" characters in position [k,i-1] and the number of other elements. When M is equal to zero, we know that in position [k,i;1] there is not any increase, thus we can eliminate it and proceed to the next portion [i+1,n].
Our objective is to prove the partial correctness of the first part of the algorithm, the most critical one. At the end of the while cycle we have to grant that, if M>0, if there is any increase is from a[k], otherwise if M=0 there is not any increase. The second part of the program is obviously correct with this precondition (and if all the element of "a" are positive).
We need to find the correct cycle invariant. We can observe that if at k-th step of the cycle a new candidate is chosen thus there is not any increase of the portion [1,k]. On the next steps, from k to i-1, M should be always positive and should represent the number of times that a[k] exceed the increase on the [k,i-1] part. Hence as cycle invariant we can choose $I = M \geq 0 \wedge k \leq i-1 \wedge \text{exceeds}(M, k, i-1) \wedge \neg m(1,k-1) \wedge 2 \leq i \leq n+1$.
The predicate m(x,y), which formalization is left to the reader, is valid, if and only if, there is a majority between elements with same indexes included between x and y. We do not formalize exceeds(M,x,y) predicate, but we state only some of its properties:

(i) $\text{exceeds}(0,x,y) \rightarrow \neg m(x,y)$
(ii) for M>0, if exceeds(M,x,y) then a[x] is a majority for the portion [x, y] and for all the portions [x,h], with $x \leq h \leq y$.

(iii) M represents the number of time by which a[x] exceed the majority, which means, if h is the multiplicity of x, M = h - ((y-x+1) div 2).

We should prove the partial correctness of the segment:

{n≥1}

```
M:=1; i:=2; k:=1;
while i≤n do
      if M=0
            then k:=1; M:=1;
            else if a[k] = a[i] then M:=M+1 else M:=M-1 fi
      fi
      i := i+1
od
```

{m(1,n) $\rightarrow$ ma(a[k]) $\wedge$ M >0}.

with ma(x) predicate which is valid, if and only if, x is a majority for the vector a.

The proof that we can derive the invariant from the precondition, which means that {n≥1} M:=1; i:=2; k:=1; {I}, is oblivious: with back word substitution of I repeated three times we obtain $1\geq 0 \wedge 1\leq 1 \wedge$ exceeds(1,1,1) $\wedge\neg$m(1,0) $\wedge$ $2\leq 2\leq n+1$, if the predicates are correctly defined becomes n≥1.

The proof that I $\wedge\neg i\geq n$ implies the post condition is the following. I $\wedge\neg i\geq n$ is equivalent to $M\geq 0 \wedge k\leq n$ $\wedge$exceeds(M,k,n) $\wedge\neg$m(1, k-1) $\wedge$ $2\leq n+1\leq n+1 \wedge i = n+1$. Given M>0. If the antecedent is true m(1,n) of the post condition, which means that exists a majority, thus should exist a value of j, j≤n, by which m(i,j) is true. Must be j≥k, because there is not any majority until k-1. Because until a “k” is found there is not a majority, thus in the portion [k,j] a[j] should be the majority. In this way there is the possibility that m(1,j) is true. But, if exceeds(M,k,n) is true, with M>0, then for the (ii) a[k] is a majority in the portion [k,j], (k≤j≤n) and then a[k] = a[j].

Because until a k there is not any majority, thus in the portion [k,j] a[j] should be the majority, in order to let m(1,j) be true. But if exceeds(M,k,n) is true, with M>0, than for the (ii) a[k] is a majority in the portion [k,j], (k≤j≤n) and then a[k] = a[j]. If M=0 then there is no majority in [1, k-1] and there is no majority in [k,n]. Thus there is no majority even in [1,n] and the post condition is verified. The last part of the demonstration should state that I is an invariant, thus, has a backward substitution:

```
{I∧ i≤n} if M=0
                then k:=1; M:=1;
                else if a[k] = a[i] then M:=M+1 else M:=M-1 fi
         fi
{$I_i^{i+1}$}
```

Which is equivalent to the followings lemmas:

(A) {I$\wedge$ i≤n $\wedge$ M=0} k:=i; M:=1; {I_i^{i+1}}

(B) {I$\wedge$ i≤n $\wedge$ M≠0} **if** a[k] = a[i] **then** M:=M+1 **else** M:=M-1 **fi** {I_i^{i+1}}.

To prove (A) we have to show that $I \wedge i \leq n \wedge M=0 \rightarrow I_{i\ M\ k}^{i+1\ 1\ i}$.

$I_{i\ M\ k}^{i+1\ 1\ i}$ is $1 \geq 0 \wedge i \leq i \wedge$ exceeds(1,i,i) $\wedge\ 2 \leq i+1 \leq n+1 \wedge \neg$m(1,i-1). First three term are true $I_{i\ M\ k}^{i+1\ 1\ i}$ is equivalent to $1 \leq i \leq n \wedge \neg$m(1,i-1)

$I \wedge i \leq n \wedge M=0$ è exceeds(0,k,i-1) $\wedge\neg$m(1,i-1) $\wedge$ k≤i-1 $\wedge$ i≤n. There is not a majority in both cases: [1,k] and [k,i-1], and thus also in [1,i-1].

We can prove the lemma (B), M≥0 when I is true, M≠0 is equivalent to M>0.

We have to prove the following lemmas, obtained from the l'**if**..**then**..**else** rule with back ward substitution:

(B1) $I \wedge i \leq n \wedge M>0 \wedge a[k] = a[i] \rightarrow I_{i\ M}^{i+1\, M+1}$

(B2) $I \wedge i \leq n \wedge M>0 \wedge a[k] \neq a[i] \rightarrow I_{i\ M}^{i+1\ M-1}$

Prove (B1). $I_{i\ M}^{i+1\ M+1}$ is:

$M+1 \geq 0 \wedge k \leq i \wedge$ exceed(M+1, k, i) $\wedge\neg$ m(1,k-1) $\wedge\ 2 \leq i+1 \leq n+1$

$I \wedge i \leq n \wedge M>0 \wedge a[k] = a[i]$ is:

$M>0 \wedge k \leq i-1 \wedge$ exceed(M, k, i-1) $\wedge\neg$ m(1,k-1) $\wedge\ 2 \leq i \leq n+1 \wedge a[k] = a[i]$.

If a[k] = a[i] and the number of element between k and i-1 is equal to a[k] then exceeds of M, the majority. From k to i the majority is exceeded of M+1 (because there is another element equal to a[k]). Other properties of exceed(M+1,k,i-1) are always true.

Prove of (B2). $I_{i\ M}^{i+1\ M-1}$ is:

$M-1 \geq 0 \wedge k \leq i \wedge$ exceeds(M-1, k, i) $\wedge\neg$ m(1,k-1) $\wedge\ 2 \leq i+1 \leq n+1$

If from k to i-1 the number of element over the majority is M>o and a[i] is not equal to a[k], then from k to i the of M-1. For M-1, k and i we know that properties of exceeds(M-1,k,i) are true. This step end the partial majority is exceeds correctness. We can observe that the demonstration is correct using a well-formed definition of the "exceeds" predicate.

Exercise 5.1.13

Consider the following bubble-sort algorithm:

```
var a: array [1..n] of integer;
begin
      i := n;
      while i>1 do
            j := 1;
            while j<i do
                  k : = j+1;
                  if a[j] > a[k]
```

```
                              then x := a[k]; a[k] := a[j]; a[j] :=
                        x; fi
                        j := j+1;
                  od
                  i := i-1;
            od
      end
```

Verify partial correctness respecting the precondition $n \geq 1$ and that the elements of *a* are distinct to each other and the post condition that *a* is an ordered permutation of the beginning vector.

Solution

We start defining the preconditions and post conditions. We use a fake array *b* to ensure that, at the end, the execution of the array *a* contains the same elements of the start array. The precondition is:

Pre = perm(a, b) $\wedge$ $n \geq 1$ $\wedge$ distinti(a)
where:

perm(a,b) = $\forall h \, (1 \leq h \leq n \rightarrow \exists m \; 1 \leq m \leq n \wedge a[h] = b[m])$

distinti(a) = $\forall h \forall m \, (1 \leq h \leq n \wedge 1 \leq m \leq n \wedge a[h] = a[m] \rightarrow h = m)$

The post condition is:

Post = perm(a,b) $\wedge$ ord(a)

with ord(a) = $\forall h \, (1 \leq h \leq n-1 \rightarrow a[h+1] > a[h])$

In the definition of the predicate *ord*, we can use a strict ordering because, in any case, every element is distinct to each other.

The program is partially correct if, given an invariant I for the external loop, we can prove the following 3 lemmas:

(L1) I $\wedge \neg$ i>1 $\rightarrow$ Post

(L3) {Pre} i:=n; {I}

(L2) I is a loop invariant

External loop invariant:

At each iteration of the external loop, the elements of the proportion [i+1, *n*] are the biggest of the array, that are arranged in ascending order: the bubble-sort algorithm moves in position *i* the maximum element in the interval [1,i], after an execution of the internal loop. Therefore, *i* can be decremented only by one and the output condition is $\neg$i>1: also $i \geq 1$ must be a part of the invariant. Concluding, at each iteration, *a* still remains a permutation of *b*. The chosen invariant is:

I = perm(a,b) $\wedge$ ord(a, i) $\wedge$ mag(a, i) $\wedge$ $i \geq 1$

where:

ord(a, i) = $\forall h \, (i \leq h \leq n-1 \rightarrow a[h] < a[h+1]])$

mag(a, i) = $\forall h \, (i+1 \leq h \leq n \rightarrow \forall m \, (1 \leq h \leq i \rightarrow a[h] > a[m]))$

ord(a,i) indicates that a[i..n] is ordered, while mag(a, i) indicates that the values in a[i+1..n] are bigger that the values contained in a[1:i].

The following properties are valid:

$ord(a) \rightarrow \forall\ i\ ((1 \leq i) \rightarrow ord(a, i))$;

$ord(a,1) \equiv ord(a)$;

ord(a,n) and mag(a, n) are tautologically true.

Proof of L1

$I \wedge \neg\ i>1 \equiv perm(a, b) \wedge ord(a, i) \wedge mag(a, i) \wedge i \geq 1 \wedge \neg\ i>1 \equiv$

$perm(a, b) \wedge ord(a, i) \wedge mag(a, i) \wedge i=1 \equiv perm(a,b) \wedge ord(a,1)$, which is precisely Post (because ord(a,1) is equal to ord(a) and ord(a) implies mag(a,1)).

Proof of L3

Considering the assignment axiom, it is enough to prove that $Pre \rightarrow I_i^n$.

$I_i^n \equiv perm(a, b) \wedge ord(a, n) \wedge mag(a, n) \wedge n \geq 1 \equiv perm(a, b) \wedge n \geq 1 \equiv Pre$.

Proof of L2

To prove that I is an invariant for the external loop, it is required to prove the following three lemmas, where J is an invariant for the internal loop:

(L2.1) $\{J \wedge \neg\ j<i\}$ i := i-1; {I}

(L2.2) $\{I \wedge i>1\}$ j := 1; {J}

(L2.3) J is an invariant for the internal loop.

Invariant J choice: as previously observed, the internal cycle of the bubble sort algorithm finds the maximum element between the first *i* and put it in position *i*. This is obtained moving the biggest elements to the top. At each iteration *j* contains the biggest element in the interval [1, j-1]. The condition on *i* remains valid because the elements in the interval [i+1, n] remains unchanged, and *a* remains a permutation of *b*, excluding that *i* is strictly bigger than 1. An invariant is:

$J = I \wedge mass(a, j) \wedge j \leq i \wedge i>1$

where $mass(a, j) = \forall h\ (1 \leq h \leq j-1 \rightarrow a[j] > a[h])$

Proof of L2.1

Applying the assignment axiom, it is enough to prove that $J \wedge \neg\ j<i \rightarrow I_i^{i-1}$.

$J \wedge \neg\ j<i \equiv I \wedge mass(a, j) \wedge j=i \wedge i>1 \equiv I \wedge mass(a, i) \wedge i \geq 2$.

$I_i^{i-1} \equiv perm(a, b) \wedge ord(a, i-1) \wedge mag(a, i-1) \wedge i-1 \geq 1$.

But $ord(a, i-1) \equiv ord(a, i) \wedge a[i-1] < a[i]$ and $mag(a,i-1) \equiv mag(a,i) \wedge \forall h\ (1 \leq h \leq i-1 \rightarrow a[i] > a[h]) \equiv mag(a,i) \wedge mass(a, i)$, and then is: $I_i^{i-1} \equiv perm(a, b) \wedge ord(a, i) \wedge a[i-1] < a[i] \wedge mag(a,i) \wedge mass(a, i) \wedge i \geq 2 \equiv I \wedge mass(a, i) \wedge i \geq 2$ (because $a[i-1] < a[i]$ is implied from mag(a,i)) $\equiv J \wedge \neg\ j<i$.

Proof of L2.2

For the assignment axiom it is enough to prove that $I \wedge i>1 \rightarrow J_j^1$.

$J^{1}_{j} \equiv I \wedge \text{mass}(a, 1) \wedge 1 \le i \wedge i>1 \equiv I \wedge i>1$ (since mass(a,1) is true), because I does not contain the variable *j*.

<u>Proof of L2.3</u>

J is an invariant if the following formula is a theorem:

$\{J \wedge j<i\}$

```
        k : = j+1;
        if a[j] > a[k] then x := a[k]; a[k] := a[j]; a[j] := x; fi
        j := j+1;
```

$\{J\}$

It is enough to prove that:

$\{J \wedge j<i \wedge k = j+1\}$ **if** a[j] > a[k] **then** x := a[k]; a[k] := a[j]; a[j] := x; **fi** $\{J^{j+1}_{j}\}$

because the backward substitution using k:=j+1 allows us to prove the theorem, because $(J \wedge j<i \wedge k=j+1)^{j+1}_{k} \Leftrightarrow J \wedge j<i$ (k not occurs in J).

Considering the **if..then** rule and the assignment axiom, it is enough to prove

for the empty else branch: (L2.3.1) $J \wedge j<i \wedge k = j+1 \wedge a[j] \le a[k] \rightarrow J^{j+1}_{j}$

for the then branch: (L2.3.2) $J \wedge j<i \wedge k = j+1 \wedge a[j] > a[k] \rightarrow J''$

where $J'' = J^{j+1}_{j}{}^{x}_{a[j]}{}^{a[j]}_{a[k]}{}^{a[k]}_{x}$.

<u>Proof of (L2.3.1)</u>

$J^{j+1}_{j} \equiv \{I \wedge \max(a,j+1) \wedge j+1 \le i \wedge i \ge 1\} = \{I \wedge \max(a,j+1) \wedge j<i \wedge i \ge 1\}$ is clearly implied by $\{J \wedge j< i \wedge k = j+1 \wedge a[j] \le a[k]\} \equiv$

$\{ I \wedge \max(a,j) \wedge j \le i \wedge i \ge 1 \wedge j< i \wedge k = j+1 \wedge a[j] \le a[j+1]\} \equiv$

$\{ I \wedge \max(a,j) \wedge i \ge 1 \wedge j< i \wedge k = j+1 \wedge a[j] \le a[j+1]\} \equiv$

$\{ I \wedge \max(a,j+1) \wedge i \ge 1 \wedge j< i \wedge k = j+1\}$

<u>Proof of (L2.3.2)</u>

Defining with "the result of the backward substitution of each predicate in J, where J'' $= J^{j+1}_{j}{}^{x}_{a[j]}{}^{a[j]}_{a[k]}{}^{a[k]}_{x}$:

$J'' \equiv \{\text{perm}''(a,b) \wedge \text{ord}''(a, i) \wedge \text{mag}''(a, i) \wedge i \ge 1 \wedge \text{mass}''(a, j) \wedge j \le i \wedge i>1\}$

Note: conditions that involve only i and j do not change, because i and j do not contain assignments in the then-branch.

We prove one predicate at a time:

We show that $\{J \wedge j<i \wedge k = j+1 \wedge a[j] > a[k]\} \rightarrow \text{perm}''(a, b)$. Naturally, perm"(a,b) = perm(a", b") = perm (a", b), because b does not change.

$a''[h] = a[h]^{j+1}_{j}{}^{x}_{a[j]}{}^{a[j]}_{a[k]}{}^{a[k]}_{x}$ = (if h=j then x else $a[h])^{a[j]}_{a[k]}{}^{a[k]}_{x}$ = (if h=j then x else if h=k then a[j] else $a[h])^{a[k]}_{x}$ = (if h=j then a[k] else if h=k then a[j] else a[h]). Hence perm"(a, b) $\Leftrightarrow \forall h\ (1 \le h \le n \rightarrow \exists m\ 1 \le m \le n \wedge a''[h] = b[m])$, that is verified under the hypothesis perm(a, b) is true. In fact, for h = j a"[h] = a[k] = a[j+1], while for h = j+1 a"[h] = a[j] and for the remaining part of values a"[h] = a[h]. For this reason a" is a permutation of *a* and *b*.

Analogously, ord"(a, i) = ∀h (i+1≤h≤n-1 → a"[h+1] > a"[h]), that is verified if ord(a, i) ∧ j<i is true, because the element with the biggest index in the array that could be changed, it is in position j+1 = i < i+1.

mag"(a, i) = ∀h (i+1≤h≤n →∀m (1≤h≤m → a"[h] > a"[m])). Because h ≥ i+1, considering, as hypothesis, i>j we have mag"(a, i) ⇔ mag(a, i), because a[j] and a[j+1] are changed and the elements from *i+1>j+1* to*n* still remain bigger that the leftmost.

mass"(a, j+1) = ∀h (1≤h≤j → a"[j+1] > a"[h]) ⇔ ∀h (1≤h≤j-1 → a[j] > a[h]) ∧ a[j] > a[j+1] ⇔ mass(a, j) ∧ a[j] > a[j+1], that is a part of the hypothesis.

Exercise 5.1.14

Consider the following problem: given a sequence x1 , .., xn of natural numbers in an arbitrary order and a sequence of integers a1 , .., an which is a permutation of 1, 2, .., n, sort the first sequence the order imposed by the permutation. For example, if x = [15, 5, 1, 9, 11] and a = [3, 2, 5, 1, 4], the order imposed by the permutation is [9, 5, 15, 1, 11], i.e. the third element in order of magnitude, followed by the second, fifth and so on. Determine an algorithm with linear time complexity that solves the problem. The algorithm must be in place, i.e. should not use another vector in addition to those given. Also prove partial correctness.

Solution

A good solution is to modify a sorting algorithm in place, such as heap sort, in order to use the comparison between the elements of A instead of those between elements of x.

For example, instead of x [i] <x [j] you compare a [i] <a [j], and "swap a[i], a[j]" is added on the instructions of the type "swap x[i], x[j]".In practice you order *a* and *x* changes accordingly. The complexity, however, is $\Theta(n \lg n)$.

There exists a better algorithm, still in place, with linear complexity. To illustrate this, we define first a formal specification for the problem addressed.

To be able to refer to the original position of the elements in *a* and in *x*, we assume that these vectors are at the beginning of computation equal to two vectors *b* and *y* respectively. Suppose also that all elements of *x* are distinct among them; we denote this assumption with the predicate distinct(x), whose formalization is left to the reader. We also define the following predicates:

ug = ∀f (1≤f≤j → x[f] = y [f] ∧ a[f] = b[f]) occurred if all the elements of *x* and *a* are in its original position.

ok(i) = ∀f (1≤f≤i →∃g b[g] = f ∧ x[f] = y[g] ∧ a[f] = f) model the relation: "the elements from 1 to i where taken in correct position".

$okug(d,e) = \forall f\ (d \le f \le e \rightarrow (\exists g\ b[g] = f \wedge x[f] = y[g] \wedge a[f] = f) \vee (x[f] = y\ [f] \wedge a[f] = b[f]))$ model the relation: "each element of *x* or *y* in the portion [d,e] is in the correct position or in the original position".

permutation(a,n) means that the elements of *a* are a permutation of 1, 2, .., n.

perm(a, b, x, y) means that *a* and *x* are obtained by permuting *b* and *y*in the same way.

A precondition is then: $ug \wedge$ permutation(a,n) $\wedge\ n \ge 0 \wedge$ distinct(x).

The post condition is that ok(n) holds and a and x are obtained by permuting b and y in the same way: ok(n) $\wedge$ perm(a, b, x, y).

The algorithm can be summarized as follows: let perm(a, b, x, y), 1□i□n, permutation(a,n), ok(i-1) and okug(i,n). If we can find a series of operations after which ok(i) and okug(i +1, n) hold, and they keep valid perm(a, b, x, y), we have obtained an algorithm based on a cycle:

i:=1; **while** i≤n **do** "series of operations"; i:=1+1; **od**

that ensures that at the end of the cycle, when i = n +1, ok(n) and perm(a, b, x, y) hold.

Observe then that, under the above assumptions, if a[i] = i, x[i] is in the correct position and then ok(i) holds, but if a[i] ≠ i, x[i] must be moved in position x[a[i]]. Then we carry out an exchange between x[i] and x[a[i]] and between a[i] and a[a[i]]. The element that was in x[i] is then in correct position, while another element was taken in i from its incorrect original position : if the correct position for this second component is exactly i, then ok(i) and okug(i, n) hold (we have not changed any other element); otherwise, it nevertheless continues to hold ok(i-1) and okug(i, n) and then you can again apply the above steps until you get in i an element whose correct position is precisely i. At the end of this sequence of operations, ok(i) holds, since elements in the positions prior to i were not changed, and ok(i +1, n) holds too, since elements in the positions next to i remain in the original position or are moved in correct position. Also, since we only carry out interchanges between pairs of corresponding elements of *x* and *a*, the predicate perm(a, b, x, y) continues to hold.

The complexity will surely be linear, since each pair of interchange leads at least one element from the original position to the correct one and then there will be at most 2n interchanges.
The program is summarized as follows:

```
{ug∧ permutation(a,n) ∧ n≥0 ∧ distinct(x)}
i:=1;
while i≤n do
    while a[i]>i do
        k:=a[i];
```

```
            t:=x[i]; x[i] := x[k]; x[k] := t;
            t:=a[i]; a[i] := a[k]; a[k] := t;
            od
      i := i+1;
od
{ok(n) ∧ perm(a, b, x, y)}
```

To prove the partial correctness of the program, we note first that the parallel execution of interchanges between pairs of items of *x* and the corresponding pair of *a* ensure that during the execution of the program and at its end is always checked perm(a, b, x , y), except that during the execution of interchanges, and therefore we ignore this condition.
The summary of the algorithm immediately suggests an invariant for the outer loop:

I = ok(i-1) ∧ okug(i,n) ∧ $1 \le i \le n+1$

In the internal cycle must be valid ok(i-1), but must be establish that $i \le n$ e a[i] $\ge$i.

In addition, the element in position *i* can be originated by a swap with an element that is moved to the correct position. It is not in its original position without being in the correct position. An invariant is:

J = ok(i-1) ∧ okug(i+1,n) ∧ $1 \le i \le n$ ∧ a[i] $\ge$i.

To prove the partial correctness, it is enough to prove the following lemmas:

(L1) $n \ge 0$ ∧permutation(a,n) $\rightarrow I_i^1$

Test:$I_i^1$is ok(0) ∧ okug(1,n) ∧ $1 \le 1 \le n+1$. But ok(0) is true for the definition of ok, while $n \ge 0$ ∧eq→ okug(1,n) ∧ $1 \le n+1$.

(L2) I ∧¬ $i \le n$ → ok(n)

Test: I ∧¬ $i \le n \Rightarrow$ I ∧ i = n+1 ⇒ ok(n)

(L3) J is an invariant of the internal cycle and implies the post conditionI_i^{i+1}.

Test:

To prove (L3), it is sufficient to prove the following three lemmas:

(L3.1) I ∧ $i \le n$ → J;

Draft of Proof: If ok(i-1) is satisfied, then a[1] =1, a[2] =2, .., a[i-1] = a[i].Because *a* is a permutation of 1, .., n, a[i], it cannot be equal to values bigger or equal than *i*. In addition, if okug(i,n) is satisfied, also okug(i+1,n) will be satisfied.

(L3.2) J ∧ ¬a[i] > i $\rightarrow I_i^{i+1}$

Draft of Proof:Given J1 = $J_{k\ \ x[k]\ x[i]\ t\ \ a[k]\ a[i]\ t}^{a[i]\ t\ \ x[k]\ x[i]\ t\ \ a[k]\ a[i]}$. We notice that ok(i-1) ∧ $1 \le i \le n$ is not modified by the backward substitution because the swaps only consider the elements of *x* and *a* in position *i* and *k≥i*. Because the elements in positions different from *i* and *k* are not modified, and in position *k* is copied the element in which the correct position is *k*, then okug(i+1, n) is satisfied. Applying the backward substitution for a[i] ≥i, we obtain:

$(a[i] \geq i)_{k\ \ x[k]\ x[i]\ t\ \ a[k]\ a[i]\ t}^{a[i]\ t\ \ x[k]\ x[i]\ t\ \ a[k]\ a[i]} \Leftrightarrow (a[i] \geq i)_{k\ \ a[k]\ a[i]\ t}^{a[i]\ t\ \ a[k]\ a[i]} \Leftrightarrow (t \geq i)_{k\ \ a[k]\ a[i]}^{a[i]\ t\ \ a[k]} \Leftrightarrow$ $(a[k] \geq i)_{k}^{a[i]} \Leftrightarrow a[a[i]] \geq i$. Then J1is equivalent to J $\wedge a[a[i]] \geq i$. In the hypothesis that J ∧ a[i] >i is satisfied, we obtain a[a[i]] >i, because a[1] =1, .., a[i-1] = i-1 and *a* is a permutation of 1, .., n.

(L3.3) {J ∧ a[i] > i} t:=x[i]; x[i] := x[k]; x[k] := t; t:=a[i]; a[i] := a[k]; a[k] := t; {J};

Draft ot Proof: I_i^{i+1} è ok(i) ∧ okug(i+1,n) ∧ 1≤i+1≤n+1, while J ∧¬a[i] > i is equivalent to ok(i-1) ∧ okug(i+1,n) ∧ 1≤i≤n ∧ a[i] = i that, in the hypothesis in which perm(a, b, x, y) is satisfied, it implies ok(i) ∧ okug(i+1,n) ∧ 1vi≤n.

Exercise 5.1.15

Given the problem of reversing an array of n elements (the first element goes in last position, the second on the second last position, etc.):

a. Provide a specification of the problem using first order formulas.
b. Implement a possible solution of the given problem using a simple programming language Algol-like.
c. Find an appropriate cycle invariant (of the cycles if the program uses more than one cycle) useful to describe the effect.
d. Prove the correctness of the program with respect of the specification using the cycle invariant of point c.

Solution point a.

Let a be the array which needs to be inverted. We will assume that the result of the inversion operation would be another array b, different from a. and that a will not be altered by the piece of code used to inverted it (which means that also its cardinality n is unchanged). The specification of the problem, using pre and post condition is the following.

{n > 0}
P
{∀ i ((1 ≤ i ≤ n) → b[i] = a[n-i+1])}

Solution point b.

The following piece of code in SMALG implements the specification given in point a.

```
h := 1;
while h ≤ n do
        b[h] := a[n-h+1];
        h := h + 1
od
```

Solution point c.

A cycle invariant for the piece of code used in point b. useful to prove the correctness is the following.

I: $\{\forall\ i\ ((1 \le i \le h\text{-}1) \rightarrow b[i] = a[n\text{-}i\text{+}1]) \wedge (h \le n + 1)\}$

Solution point d.

d.1 Back propagation I through h := 1 gives TRUE because for h = 1 the antecedent of the first part I is false for any i; furthermore $n > 0$ implies $1 \le n + 1$.

d.2 $I \wedge h > n$ implies $h = n + 1$ and so the post condition desired.

d.3 We will now demonstrate that I is a cycle invariant.
The clause $h \le n\text{+}1$, back propagated through h := h+1 produces $h \le n$ which is implied by the condition of the cycle (the first instruction of the cycle does not change neither h nor n). Lets focus our attention on the main clause I':$\forall\ i\ ((1 \le i \le h\text{-}1) \rightarrow b[i] = a[n\text{-}i\text{+}1])$. Back propagating I' through the body of the cycle we will obtain
I*: $\forall\ i\ ((1 \le i \le h) \rightarrow (\textbf{if}\ i\text{=}h\ \textbf{then}\ a[n\text{-}i\text{+}1]\ \textbf{else}\ b[i]) = a[n\text{-}i\text{+}1])$
I imply I*. In fact with i between 1 and h-1 the assertion of I* is coincides with the assertion of I', while with i=h the right clause of the implication is reduced at a a[n-i+1] = a[n-i+1] which is true.

Exercise 5.1.16

Given a set of integers stored in an array, possibly with repetitions.

1. Specify, by appropriate pre- and post-conditions, a portion of a program that eliminates (completely) an element from such a set.
2. Build a portion of the program that implements such erase operation.
3. Prove the correctness of the implementation built at point 2, with respect to the specifics given at point 1.
4. Prove the termination of the program constructed at point 2.

Solution

Premise

As in most programming languages, arrays have fixed sizes while the cardinality of the sets they represent may vary, so you should first establish how an array A of n elements represents a set S of cardinality card (assuming a priori card $\le$ n). Several different choices are possible:

A. The elements of S are stored in A in a spread way. The positions of A in which there is no element of S are marked by a '*' (we neglect any problems caused by the type rules of the programming language adopted).
B. The elements of S are stored in the first h positions of A, card $\le h \le n$, while in the remaining n-h positions we find '*'. So, both in case A. that in case B.

   $x \in S \leftrightarrow (\exists i\ ((1 \le i \le n) \wedge (A[i] = x)$.

   Note also that in case B. the following property must always hold in order that A is a correct representation of a set.

   $(\bullet\bullet)(\exists h\ ((0 \le h < n) \wedge (\forall i\ ((1 \le i \le h) \rightarrow (A[i] \ne *)) \wedge (\forall i\ (h < i \le n) \rightarrow (A[i] = *)))$

C. The elements of S are stored in the first h positions of A, card $\leq$ h $\leq$ n. Furthermore, we keep the value of h stored in an appropriate variable. This makes it unnecessary to explicitly assign a value to the elements of A whose index is greater than h. In this case

$(x \in S \leftrightarrow (\exists i ((1 \leq i \leq h) \wedge (A[i] = x)) \wedge (h \leq n)$.

Clearly, A., B. and C. lead to solutions to the problem that differ each other for difficulty of implementation and analysis, and for efficiency. We'll discuss here solution A. that allows a very simple solution of the exercise, looking in a very short way the other cases.

Case A

Solution point 1

The specification of a program fragment P, which eliminates the element e from a set S represented by the array A is as follows.

$\{(n > 0)\}$

P

$\{(\forall i\ (1 \leq i \leq n) \rightarrow (A[i] \neq e))\}$

(Note that in this way we ask explicitly only that, at the end of P, the element e is not present anymore in the set S represented by A. We does not encode the requirement, perhaps implicit in the enunciation of the exercise, that the other elements of S must remain in it and that cannot be added any other spurious element)

Solution point 2

A fragment of a program that eliminates all occurrences of e from S (without affecting the other elements of S) is as follows.

```
begin
    i := 1;
    while i ≤ n do
        if A[i] = e then A[i] := * fi;
        i := i + 1
    od
end
```

Solution point 3

The partial correctness of P with respect to the specifics can be proved by the following invariant:

I: $\{(\forall j\ (1 \leq j < i) \rightarrow (A[j] \neq e)) \wedge (i \leq n + 1)\}$

Indeed:

5. $I \wedge (i > n)$ implies $i = n + 1$ and then the desired postcondition.
6. Backpropagating I through i: = 1 we obtain $1 \leq n + 1$ (the first part of I becomes true in a trivial way because its antecedent is false) that is immediately implied by prerequisite.
7. I is actually an invariant of the loop. In fact, named I*

$\{(\forall j\ (1 \le j \le i) \to (A[j] \neq e)) \wedge (i \le n)\}$ obtained backpropagating I through i: = i +1;

1. $I \wedge (A[i] \neq e)) \wedge (i \le n)$ implies I*
1. Named I^ the formula

   $\{(\forall j\ (1 \le j \le i) \to ((\textbf{if}\ j = i\ \textbf{then}\ *\ \textbf{else}\ A[j]) \neq e)) \wedge (i \le n)\}$

   obtained backpropagating I* through A[i] := *

   $I \wedge (A[i] = e)) \wedge (i \le n)$ implies I^.

   In fact $(A[j] \neq e)$ is implied by I for all the values of $j < i$; while for $j = i$ I* requires $* \neq e$ that is obviously true, because * isn't en an acceptable element of S.

Solution point 4

The termination of P is obvious, given the clear equivalence of the loop with a **for** loop.

Case B

Solution point 1

The specification of a program fragment P, which eliminates the element e from a set S represented by the array A according to the scheme of representation B. is as follows:

$\{(n > 0) \wedge (\exists h\ ((0 \le h < n) \wedge (\forall i\ ((1 \le i \le h) \to (A[i] \neq *)) \wedge ((h < i \le n) \to (A[i] = *)))\}$

(In this way we ensure that A represents a set, according to the conventions established previously. Moreover S is empty if and only if h = 0)

P

$\{(\forall i\ (1 \le i \le n) \to (A[i] \neq e)) \wedge$

$(\exists h\ ((1 \le h < n) \wedge (\forall i\ ((1 \le i \le h) \to (A[i] \neq *)) \wedge (\forall i\ (h < i \le n) \to (A[i] = *)))\}$

(Note that, as in case A., in this way we ask explicitly only that, at the end of P, the element e is no longer in the set S represented by A. Instead, the required property must hold to secure that A is actually a representation of a set)

Solution 2.

A fragment of a program that eliminates all occurrences of e from S (without affecting the other elements of S) is as follows.

```
begin
i := 1;
while A[i] ≠ '*' do
```

```
        if A[i] ≠ e
        then i : = i + 1
        else
           j := i;
           while A[j] ≠ '*' do
              A[j] := A[j+1]; j := j+1
           od
        fi
    od
    end
```

Solution 3

The correctness analysis of this program, of quadratic complexity in the worst case, needs the introduction of two invariants. The outer loop invariant has to formalize the fact that, up to the index i (excluded), A does not contains occurrences of e (as well as to maintain valid the (••), that ensures that A is a correct representation of a set); so that, at the end of the loop, considering that from A[i] onwards all the elements of A are *, the required postcondition will be valid.

The invariant is not required for the inner loop (you just need to add in the outer loop invariant, the predicate $j \geq i$, that can demonstrate immediately that the outer invariant is maintained by the inner loop).

The apparent "futility" of the inner loop is explained by the weakness of the specification that does not impose that the content of A remains unchanged - excluding the cancellation of e - after the execution of the program: if this requirement was formalized by the specification, both invariants should have been "hardened" consequently.

Solution 4

We can demonstrate the termination of both loops using the (••).

The solution of the case **C.** can be developed in a similar way of the solution **B.**, updating the variable h that now becomes a program variable.

So, the variable h will be free in the specification and in the other proof of correctness formulas.

Exercise 5.1.17

We have some set of integers stored using array, possibly with repetitions.

1. Specify, by appropriate pre-and post-conditions, a portion of a program that, given two sets, compute their intersection.
2. Build a portion of the program that implements such intersection operation.
3. Prove the correctness of the implementation built in section 2 with respect to the specification given in 1.

Solution

Also in this case, as in Exercise 5.1.16, both the specification and its implementation depend on the technique used to represent a set by means of an array.

For example, we can assume that a generic set of integers S is represented by an array A of n elements and that an element belongs to S if and only if it is also an element of the corresponding array. Namely:
$x \in S \leftrightarrow \exists\, i\, ((1 \le i \le n) \wedge A[i] = x)$

In this case, said A1 and A2 the arrays that represent S1 and S2 respectively, the specification by pre- and post-conditions of a program fragment P that computes the array C representing the intersection of S1 and S2 is formalized as follows , assuming a priori that P does not change A1 and A2.

$\{n \le 0\}$
P
$\{\forall\, x\, \big(\exists i\, ((1 \le i \le n) \wedge C[i]=x) \leftrightarrow \big(\exists i\, ((1 \le i \le n) \wedge A1[i]=x) \wedge \exists i\, ((1 \le i \le n) \wedge A2[i] = x)\big)\big)\}$

In this case we will use instead the technique of representation of the sets adopted in the solution C. of Exercise 5.1.16, i.e. the elements of S are stored in the first h positions of A, card $\le h \le n$.

Furthermore, we store the value of h in an proper variable. This makes it unnecessary to explicitly assign a value to the elements of A whose index is greater than h. In this case

$(x \in S \leftrightarrow (\exists i\, ((1 \le i \le h) \wedge (A[i] = x)) \wedge (h \le n).$

Solution 1

Let A1 and A2 be the arrays representing the two sets to intersect. Let A3 be the array that will contain the result. They are associated, respectively, to the variables h1, h2 and h3. The physical dimensions of the array are n for each of them. The specification of an intersection program is therefore the follow.

$\{h1 \le n \wedge h2 \le n\}$
INTERSECTION
$\big\{h3 \le n \wedge$

$\big(\forall i\, ((1 \le i \le h3) \rightarrow (\exists j,r\, ((1 \le j \le h1) \wedge (1 \le r \le h2) \wedge (A3[i] = A1[j] = A2[r])))\big) \wedge$

$\big(\forall j,r\, ((1 \le j \le h1) \wedge (1 \le r \le h2) \wedge (A1[j] = A2[r])) \rightarrow$
$(\exists i\ (1 \le i \le h3) \wedge (A3[i] = A1[j])))\big)\big\}$

Solution 2
An implementation of the program INTERSECTION is:
```
begin
i1 := 1; i3 := 1;
while i1 ≤ h1 do
```

```
        i2 := 1; found := false;
        while i2 ≤ h2 and not found do
            if A1[i1] = A2[i2] then found := true else i2 := i2 + 1 fi
        od;
        if found then A3[i3] := A1[i1]; i3 := i3 + 1 fi
        i1 := i1 + 1
    od
    h3 := i3 - 1
    end
```

Solution 3

The proof of correctness of the program INTERSECTION respect to its specification can be constructed through the following steps:

- Invariant of the outer loop:
  IEXT:

  $\{(i3 \le i1 \le h1 + 1) \wedge$
  $(\forall i\ ((1 \le i \le i3 - 1) \rightarrow$
  $(\exists j,r\ ((1 \le j \le i1 - 1) \wedge (1 \le r \le h2) \wedge (A3[i] = A1[j] = A2[r]))) \wedge$
  $(\forall j,r\ ((1 \le j \le i1 - 1) \wedge (1 \le r \le h2) \wedge (A1[j] = A2[r])) \rightarrow$
  $(\exists i\ (1 \le i \le i3 - 1) \wedge (A3[i] = A1[j]))\}$

  IEXT provides that, at each iteration of the loop, A3 contains all and only the items that are in the portion of A1 whose indexes are between 1 and i1 – 1, and in A2.

- The proof that IEXT ensures the postcondition at the cycle exit(and after the execution of h3: = i3 - 1) and that IEXT is guaranteed by the program initialization is immediate.
- The back propagation of IEXT through i1 := i1 + 1 produces the following formula:
  (*)
  $\{(i3 \le i1 + 1 \le h1 + 1) \wedge$
  $(\forall i\ ((1 \le i \le i3 - 1) \rightarrow$
  $(\exists j,r\ ((1 \le j \le i1) \wedge (1 \le r \le h2\ ) \wedge (A3[i] = A1[j] = A2[r])))$
  $\wedge$
  $(\forall j,r\ ((1 \le j \le i1) \wedge (1 \le r \le h2) \wedge (A1[j] = A2[r])) \rightarrow$
  $(\exists i\ (1 \le i \le i3 - 1) \wedge (A3[i] = A1[j]))\}$
  The back propagation of (*) through the then branch of the conditional instruction yields:
  (**)
  $\{(i3 \le i1 \le h1) \wedge$
  $(\forall i\ ((1 \le i \le i3) \rightarrow (\exists j,r\ ((1 \le j \le i1) \wedge (1 \le r \le h2) \wedge$
  $((\textbf{if}\ i = i3\ \textbf{then}\ A1[i1]\ \textbf{else}\ A3[i]) = A1[j] = A2[r])))$
  $\wedge$
  $(\forall j,r\ ((1 \le j \le i1) \wedge (1 \le r \le h2) \wedge (A1[j] = A2[r])) \rightarrow$
  $(\exists i\ (1 \le i \le i3) \wedge ((\textbf{if}\ i = i3\ \textbf{then}\ A1[i1]\ \textbf{else}\ A3[i]) = A1[j]))\}$
  Now consider the following formula:
  (***)

$\{(i3 \leq i1 \leq h1) \wedge (i2 \leq h2 + 1) \wedge$
$(\forall i\ ((1 \leq i \leq i3 - 1) \rightarrow$
$\quad(\exists j,r\ ((1 \leq j \leq i1 - 1) \wedge (1 \leq r \leq h2) \wedge (A3[i] = A1[j] = A2[r]))) \wedge$
$(\forall j,r\ ((1 \leq j \leq i1 - 1) \wedge (1 \leq r \leq h2) \wedge (A1[j] = A2[r])) \rightarrow$
$\qquad(\exists i\ (1 \leq i \leq i3 - 1) \wedge (A3[i] = A1[j])) \wedge$
$found \leftrightarrow A1[i1] = A2[i2 - 1] \wedge$
$\neg\ found \leftrightarrow (\forall r\ (1 \leq r \leq h2) \rightarrow (A1[i1] \neq A2[r]))$
The (***) states that all values of A3 whose indexes are between 1 and i3 - 1 can also be found in A2 and in the A1 portion between 1 and i1 - 1 and vice versa.
Besides, the variable found is true if and only if A1 [i1] = A2 [i2 - 1] while, if it is false, the value of A1 [i1] is not in A2.
Observe now that (***) ∧found implies (**) while (***)∧ ¬ found implies (*). To verify this just consider the cases j = i3 = i1 in both parts of (*) and (**). Consider for example the subformula
$(\forall i\ ((1 \leq i \leq i3) \rightarrow (\exists j,r\ ((1 \leq j \leq i1) \wedge (1 \leq r \leq h2) \wedge$
$\qquad((\textbf{if}\ i = i3\ \textbf{then}\ A1[i1]\ \textbf{else}\ A3[i]) = A1[j] = A2[r])))$

of the (**). For i = i3 must exist j,r ((1 ≤ j ≤ i1) ∧ (1 ≤ r ≤ h2) such that A1[i1] = A1[j] = A2[r]. This is guaranteed by A1[i1] = A2[i2 - 1] (implied by found) and by i2 ≤ h2 + 1. The other cases can be treated similarly.

We can therefore conclude that

```
{(***)}
if found then A3[i3] := A1[i1]; i3 := i3 + 1 fi
{(*)}
```

- It therefore remains to be demonstrated that (***) holds the inner loop exit. The goal is easily done by taking as invariant the (***), just replacing h2 with i2 – 1 in the subformula $\neg found \leftrightarrow (\forall r\ (1 \leq r \leq h2) \rightarrow (A1[i1] \neq A2[r]))$ and observing that at the loop exit the condition (i2 > h2 or found) holds.

5.2 Specification and proof of systems' properties

Exercise 5.2.1

Question a

Formalize, using first-order logic formulas, the following system. A floodgate in a hydroelectric reservoir is controlled by the following signals: *up* to open it and *down* to close it.

The floodgate may be in one of the following states: *up, down, movingUp and movingDown.*

When the floodgate is in a state □{*up, down*} and receives a command that is the opposite of its actual state, the floodgate starts moving and it arrives to the opposite state after Δ time unit. If it receives a command consistent with the direction of its movement while it is moving, the command is ignored, but if the command is opposite to the direction of movement (e.g. the floodgate is *movingDown* and it receives *up*) it takes effect only after the floodgate reaches the previous state.

Model the system using the following predicates:

Position: binary, having as object the state of the floodgate and the state's instant of validity

Command: binary, having as object the given command and the instant when it's given.

Question b (*)

Verify if the specific action built in question a. ensures (logically implies) the following property: when the floodgate receives any command, it reaches the required state within 2Δ time units.

Solution a

❑ The following formula formalizes that, while the floodgate is in *up* position during a generic instant t and it receives a *down* command, it moves in the *movingDown* state for Δ time units. Then it moves to the *down* state.

$$\Box t \text{ position}(up, t) \Box \text{ go}(down, t) \rightarrow$$
$$((\Box t_1 \ (t < t_1 < t + \Delta) \rightarrow \text{position}(movingDown, t_1)) \Box \text{ position}(Down, t+\Delta))$$

2. The following formula formalizes that, while the floodgate is in *movingDown* position during a generic instant t and it receives a *up* command, it remains in the *movingDown* state for a certain period of time (it's not necessary to specify the duration of this interval because the previous formula implies that when the floodgate begins a movement it has to take exactly 5 time units). Then it moves to the *down* state and immediately to the *movingUp* state that takes exactly Δ time units before ending to the *up* state.

$$\Box t \text{ position}(movingDown, t) \Box \text{ go}(up, t) \rightarrow$$
$$(\Box t_2 \ ((\Box t_1 \ (t < t_1 < t_2)^3 \rightarrow \text{position}(movingDown, t_1)) \Box \text{ position}(Down, t_2) \Box$$
$$(\Box t_3 \ (t_2 < t_3 < t_2 + \Delta) \rightarrow \text{position}(movingUp, t_3)) \Box \text{ position}(up, t_2 + \Delta))$$

3. 4. Symmetrical formulas of 1 and 2 specify the opposite movements of the floodgate

❑ The following formula establishes that the floodgate is always in one and only one state among the 4 possible.

3 One can show that $t2 < t + \Delta$.

$\square t\ (\text{position}(x,t)\ \square\ ((x = \textit{up})\ \square\ (x = \textit{down})\ \square\ (x = \textit{movingDown}\ )\ \square\ (x = \textit{movingUp}\ ))\ \square\ \ \square x, y\ (x \neq y \rightarrow \neg\ ((\text{position}(x, t)\ \square\ (\text{position}(y, t))$

Solution b

The desired property is stated as below:

(~) $\square t\ (\text{go}(\textit{down}, t) \rightarrow (\square t_1\ (t \leq t_1 < t + 2^*\Delta)\ \square\ \text{position}(\textit{down}\ , t_1)))$

A symmetric formula can be used for command up.

Such property is not certified by the specification formalized by 1-4. In fact a system could behave as the schema in the image 5.2 without violating any requirement stated in 1-4.

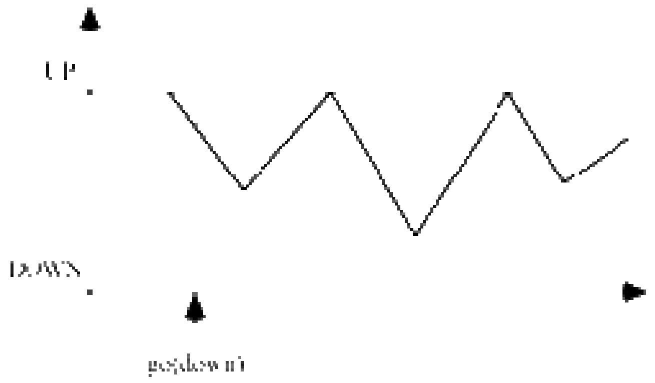


Figure 5-2 A possible graph for the floodgate movement.

As shown in Figure 52, the floodgate initially is in *up* state; spontaneously (!) starts a movement movingDown (this possibility is not excluded by 1-4); during this movement receive the command go(*down*); but nothing stated in 1-4 forces the system to reach the down state: the floodgate could always stop and return to high state. Receiving the command during the movingUp state, would force the floodgate to complete it, after that, down state will be reached.

Advanced

Adding some specifications to the floodgate movement in order to obtain the desired property, we have to admit that the previous properties are incomplete because they do not define the law of the movement of the floodgate. We have to complete those laws of the working process of the floodgate. This could be done in the following way:

1. The following formula states that it is not possible to give simultaneously the command up and the command down.

$\square t \neg\ (\text{go}(\textit{up}, t)\ \square\ \text{go}(\textit{down}, t))$

2. The following formula states that if in a generic moment the floodgate is moving up, then it is necessarily depending on the fact that less than Δtime

unit ago the position was down and that a go(up) command was given or that less than 2*Δmoment before the position was movingDown and a go(up) command was given.

$\Box$t position(*movingUp* , t) → ((($\Box$$t_1$ (t-Δ ≤ t_1 < t) $\Box$ position(*down* , t1)$\Box$ go(*up* , t_1)) $\Box$ (($\Box$$t_1$ (t-2Δ ≤ t_1 < t) $\Box$ position(*movingDown* , t_1) $\Box$ go(*up* , t_1)))

3. A symmetric formula of 7 is associated with the movingDown state.
4. The following formula formalizes the fact that the floodgate could change state from down to up only through the movingUp state and that the elapsed time must be Δ time unit

$\Box$$t_1$, t_2 ((position(*down*, t_1) $\Box$ position(*up*, t_2) $\Box$ (t_2>t_1)) → (($\Box$$t_3$, t_4 (t_4= t_3+Δ) $\Box$ (t_1<t_3<t_4<t_2) $\Box$ (position(*down*, t_3) $\Box$ position(*up*, t_4) $\Box$ $\Box$$t_5$ (t_3<t_5<t_4) →position(*movingUp*, t_5)

5. A formula like 9. describes the symmetric case.

At this time, the desired property (~) could be demonstrated as a consequence of the 1-10 as follows.

A. From 5, at any t the position assumes one of the 4 fixed values. So, if we demonstrate that:

($\Box$) $\Box$t position(*x*, t) $\Box$ go(*down*, t) → ($\Box$$t_1$ (t ≤ t_1 < t + 2*Δ) $\Box$ position(*down* , t_1))

is true for each value of x in {*up, down, movingUp, movingDown*}, the (~) will be demonstrated. We'll examine each case separately:

A.1 If x = *up*, (*) is an obvious consequence of the 1. (setting t1 = t + Δ)

A.2 If x = *down* (*) can be proved simply by setting t1 = t.

A.3 If x = *movingDown*, (*) is an immediate consequence of the following Lemma1 (used also in case A.4)

Lemma 1

$\Box$t position(*movingDown* , t) →

($\Box$$t_2$ ((t < t_2 < t + Δ) $\Box$ ($\Box$$t_1$ (t < t_1 < t_2) → position(*movingDown* , t_1)) $\Box$

position(*Down* , t_2))

Lemma1 can be proved as a consequence of the rule 8. In fact we can prove that one of the following statements is true considering such rule and the Lemma1 hypothesis (the floodgate is in movement from up to down):

B.1 ((($\Box$$t_5$ (t - Δ ≤ t_5 < t) $\Box$ position(*up* , t_5)$\Box$ go(*down* , t_5))

{variable identifiers are changed in order to avoid confusion}

B.2 (($\Box$$t_6$ (t - 2*Δ ≤ t_6 < t) $\Box$ position(*movingUp* , t_6) $\Box$ go(*down* , t_6)))

In case B.1 the Lemma1 thesis is derived from rule1, equating variable t5 of B.1 to t of 1 and retrieving the value t_5 + Δ as value of the t2 variable, used in Lemma1

In the B.2 case, the thesis is derived in a similar way, as in rule 4.

A.4 If x = movingUp, the thesis is an immediate consequence of the 4. and of the Lemma1 that qualifies, in an easier and more accurate way, variable t_2 of 4, specifying that $t_2 < t + \Delta$. t_1

Exercise 5.2.2

Question a

Specify, using first-order formulas, the behaviour of an automatic level crossing that it has to step down when a train passes through it. The level crossing receives a signal from a sensor, placed at a distance *d* form the crossroads.

The crossroads has a length *l*. V_{min} and V_{max} are, respectively, the train minimum and maximum velocity (known a priori). In addition, the level crossing has to step up when the train leaves the crossroads. The Figure 5-3 shows the desired mechanism.

Question b (*)

Specify, using first-order formulas, the property that the level crossing is always closed when a train crosses the crossroads. Demonstrate that the level crossing ensures the desired property.

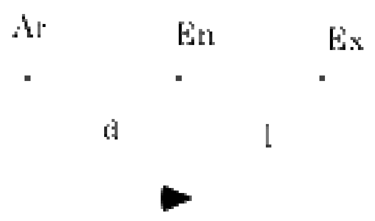


Figure 5-3 Level crossing path (monodirectional).

Suggestions.

You can assume the following simplifying hypotheses:

1. There is only one train track and trains proceed along one direction.
2. There is a distance between two trains that ensures that a train signals its arrival only after the previous train has left the crossroad (You can formalise the crossroad considering only the arrival of one train).
3. The movement of the bars is instantaneous.

You are invited to use the following predicates:

- Ar: true when train is sighted, so the signal is sent.
- En: true when the train enters the crossroads.
- Ex: true when the train moves out of the crossroads.

- Up and Down: they specify the position of the barriers.

The level crossing specifics will consist of formulas that indicate when the bar should be lowered
depending on the signal Ar.

Solution a

It is clear that the barriers should be lowered, for security reasons, when time $\delta_{min} = d/V_{max}$ is passed from the arrival of the train (Ar). Subsequently, the bar should be lifted up after a time $\delta_{max} = (d + l)/V_{min}$.

This is formalized in the following way:

α) $\square t\ (Ar(t) \rightarrow (\square t_1\ ((t+\delta_{min} \le t_1 \le t+\delta_{max}) \rightarrow Down(t_1))) \square$

$\square t\ (Down(t) \rightarrow \square t_1\ ((t-\delta_{max} \le t_1 \le t-\delta_{min})\ \square\ Ar(t_1))) \square$

$\square t\ (Down(t) \leftrightarrow Up(t))$

Solution b

Define the predicate Inc that states that the train is in the crossroad.
The following axiom is consequently true:

$(\delta_1=d/V_{min}\ \square\ \delta_2=(d+l)/V_{max}\ \square\ \delta_3=l/V_{min}\ \square\ \delta_4=l/V_{max})\ \square$

$\square t\ (Ar(t) \rightarrow \square t_1\ (En(t_1)\ \square\ (t+\delta_{min} \le t_1 \le t+\delta_1))\ \square\ \square t_2\ (Ex(t_2)\ \square\ (t+\delta_2 \le t_2 \le t+\delta_{max})))\ \square$

$\square t\ (En(t) \rightarrow \square t_1\ (Ar(t_1)\ \square\ (t-\delta_1 \le t_1 \le t-\delta_{min})))\ \square$

$\square t\ (Ex(t) \rightarrow \square t_1\ (Ar(t_1)\ \square\ (t-\delta_{max} \le t_1 \le t-\delta_2)))\ \square$

$\square t\ (Inc(t) \leftrightarrow \square t_1\ (En(t_1)\ \square\ (t-\delta_3 \le t_1 \le t-\delta_4))\ \square\ \square t_2\ (Ex(t_2)\ \square\ (t+\delta_4 \le t_2 \le t+\delta_3)))$

The desired security property is formalized as follows:

(*) $\square t\ (Inc(t) \rightarrow Down(t))$

(*) can be demonstrated as follows:

Inc(t) implies $\square t_1\ (En(t_1)\ \square\ (t-\delta_3 \le t_1 \le t-\delta_4))$.

Such statement implies that:

$\square t_2\ (Ar(t_2)\ \square\ (t-\delta_1-\delta_3 \le t_1 \le t-\delta_4-\delta_{min}))$

Since $\delta_{min} \le \delta_{min} + \delta_4 \le \delta_1 + \delta_3 \le \delta_{max}$, according to specific action α), the thesis follows.