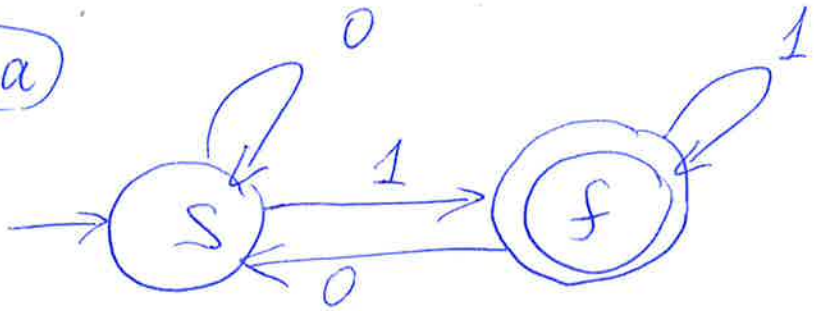
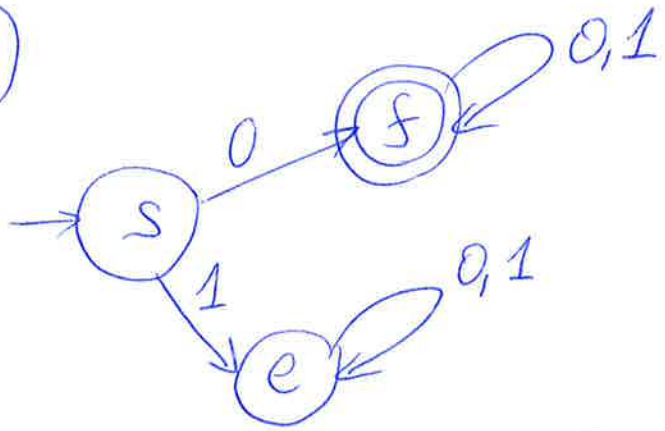
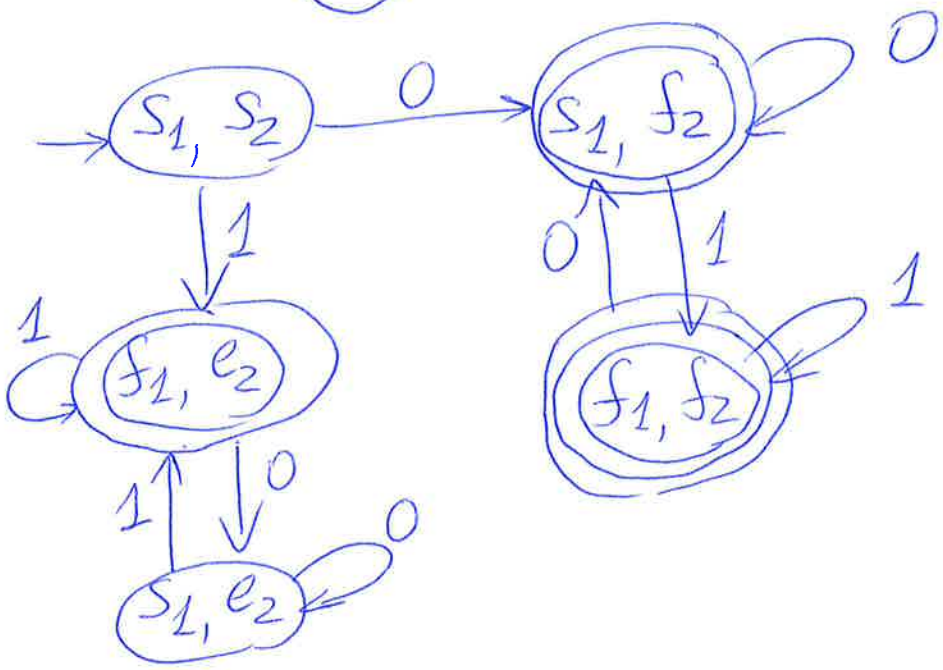
1a



1	0	1
s	f	s	f

1b





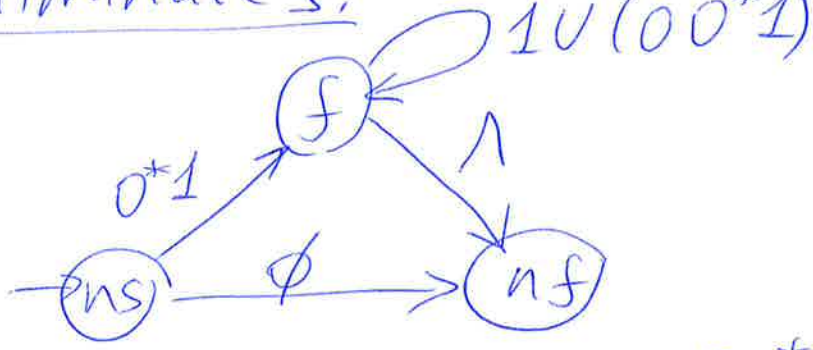
1c

x	y		z	
	1	0	1	
s	f		s	f

$x = \Lambda$
$y = 10$
$z = 1$

1d

Eliminate s:



$$R'_{ns,f} = R_{ns,f} \cup (R_{ns,s} R_{s,s}^* R_{s,f}) =$$
$$\emptyset \cup (\wedge \; 0^* \; 1) = 0^*1$$

$$R'_{ns,nf} = R_{ns,nf} \cup (R_{ns,s} R_{s,s}^* R_{s,nf}) =$$
$$\emptyset \cup (\wedge \; 0^* \; \emptyset) = \emptyset$$

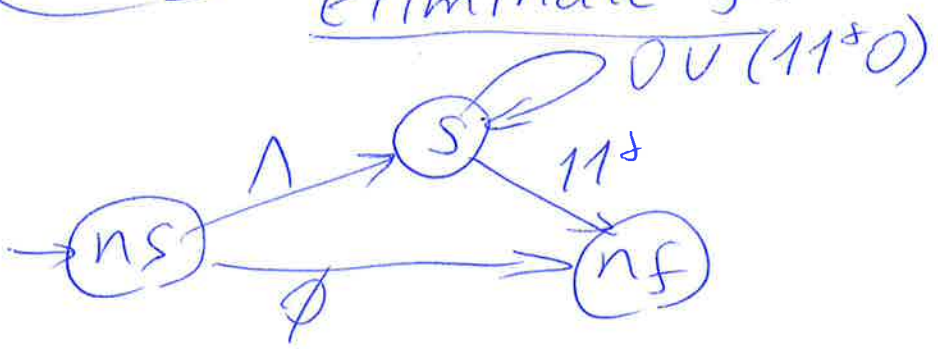
$$R'_{f,f} = R_{f,f} \cup (R_{f,s} R_{s,s}^* R_{s,f}) =$$
$$1 \cup (0 \; 0^* \; 1)$$

$$R'_{f,nf} = R_{f,nf} \cup (R_{f,s} R_{s,s}^* R_{s,nf}) =$$
$$\wedge \cup (0 \; 0^* \; \emptyset) = \wedge$$

$$R'_{ns,nf} = R_{ns,nf} \cup (R_{ns,f} R_{f,f}^* R_{f,nf}) =$$
$$\emptyset \cup (0^*1 \; (1 \cup (00^*1))^* \wedge) =$$
$$0^*1 \; (1 \cup (00^*1))^*$$

(d) cont-d

Eliminate f:

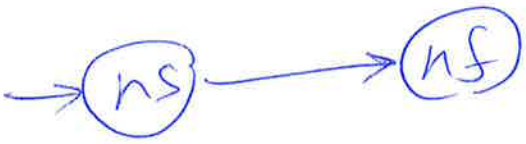


$$R'_{ns,s} = R_{ns,s} \cup (R_{ns,f} R_{f,f}^{*} R_{f,nf}) =$$
$$\Lambda \cup (\emptyset \ \dots \ ) = \Lambda$$

$$R'_{ns,nf} = R_{ns,nf} \cup (R_{ns,f} R_{f,f}^{*} R_{f,nf}) =$$
$$\emptyset \cup (\emptyset \ \dots \ ) = \emptyset$$

$$R'_{s,s} = R_{s,s} \cup (R_{s,f} R_{f,f}^{*} R_{f,s}) =$$
$$0 \cup (1 \ 1^{*} \ 0)$$

$$R'_{s,nf} = R_{s,nf} \cup (R_{s,f} R_{f,f}^{*} R_{f,nf}) =$$
$$\emptyset \cup (1 \ 1^{*} \ \Lambda) = 11^{*}$$

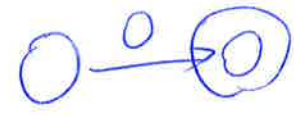


$$R'_{ns,nf} = R_{ns,nf} \cup (R_{ns,s} R_{s,s}^{*} R_{s,nf}) =$$
$$\emptyset \cup (\Lambda (0 \cup (11^{*}0))^{*} 11^{*}) =$$

$$(0 \cup (11^{*}0))^{*} 11^{*}$$

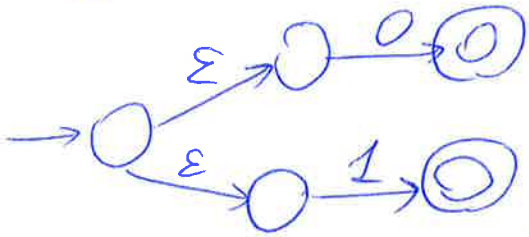
1ef

0

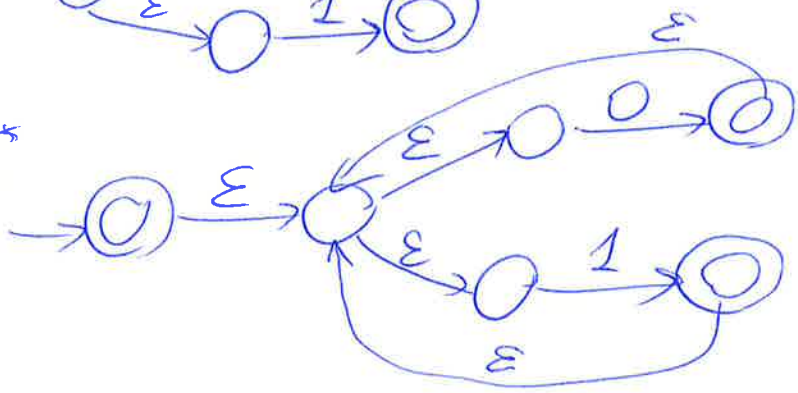


1

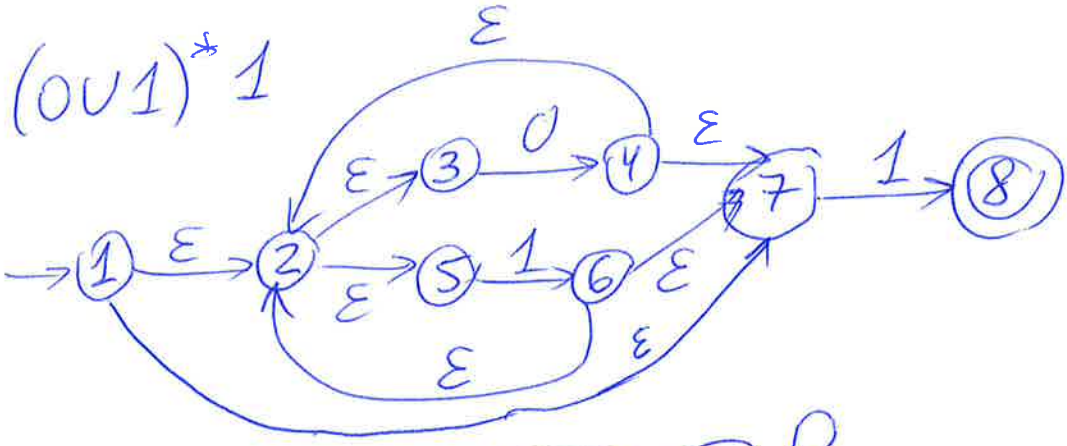
0 ∨ 1

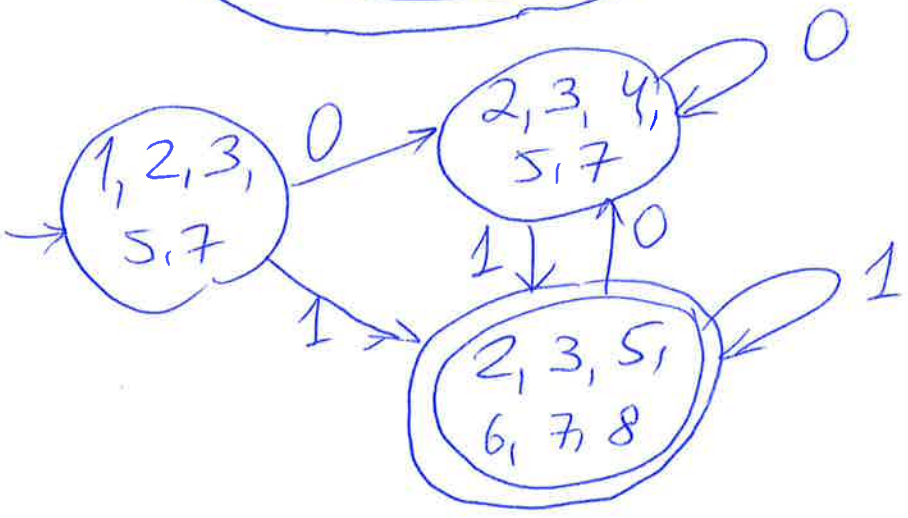


$(0 \vee 1)^*$



$(0 \vee 1)^* 1$





2a. We will prove it by contradiction. Let us assume that the given language L is regular.

Since this language is regular, according to the Pumping Lemma, there exists an integer p such that every word from L whose length $\text{len}(w)$ is at least p can be represented as a concatenation $w = xyz$, where:

- y is non-empty;
- the length $\text{len}(xy)$ does not exceed p, and
- for every natural number i, the word $xy^iz \stackrel{\text{def}}{=} xy\ldots yz$, in which y is repeated i times, also belongs to the language L.

Let us take the word

$$w = 0^p 1^{p+1} = 0\ldots 01\ldots 1,$$

in which first 0 is repeated p times, then 1 is repeated $p+1$ times. The length of this word is $p + p + 1 = 2p + 1 > p$. So, by pumping lemma, this word can be represented as $w = xyz$ with $\text{len}(xy) \le p$. This word starts with xy, and the length of xy is smaller than or equal to p. Thus, xy is among the first p symbols of the word w – and these symbols are all 0's. So, the word y only has 0's.

Thus, when we go from the word $w = xyz$ to the word $xyyz$, we add 0's, and we do not add any 1's. So, in the word $xyyz$, there are more than p 0s – i.e., at least $p+1$ zeros, and still $p+1$ 1s. Thus, in the word $xyyz$, we no longer have more 1s than 0s, so this word cannot be in the language L.

On the other hand, by Pumping Lemma, the word $xyyz$ must be in the language L. So, we proved two opposite statements:

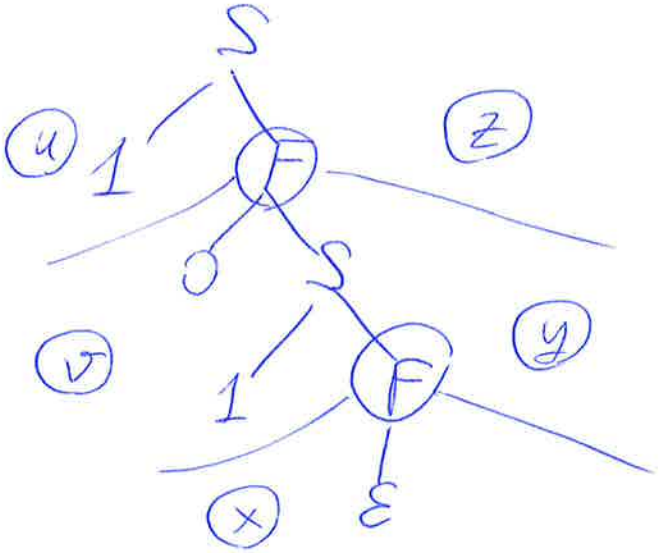
- that this word *is not* in L and
- that this word *is* in L.

This is a contradiction.

The only assumption that led to this contradiction is that L is a regular language. Thus, this assumption is false, so L is not regular.

2b, 2c

$S \to 0S$
$S \to 1F$
$F \to 0S$
$F \to 1F$
$F \to \varepsilon$



$u = 1, \quad v = 01, \quad x = y = z = \varepsilon$

2d

~~$S \to 0S$~~
~~$S \to 1F$~~
~~$F \to 0S$~~
~~$F \to 1F$~~
$F \to \varepsilon$

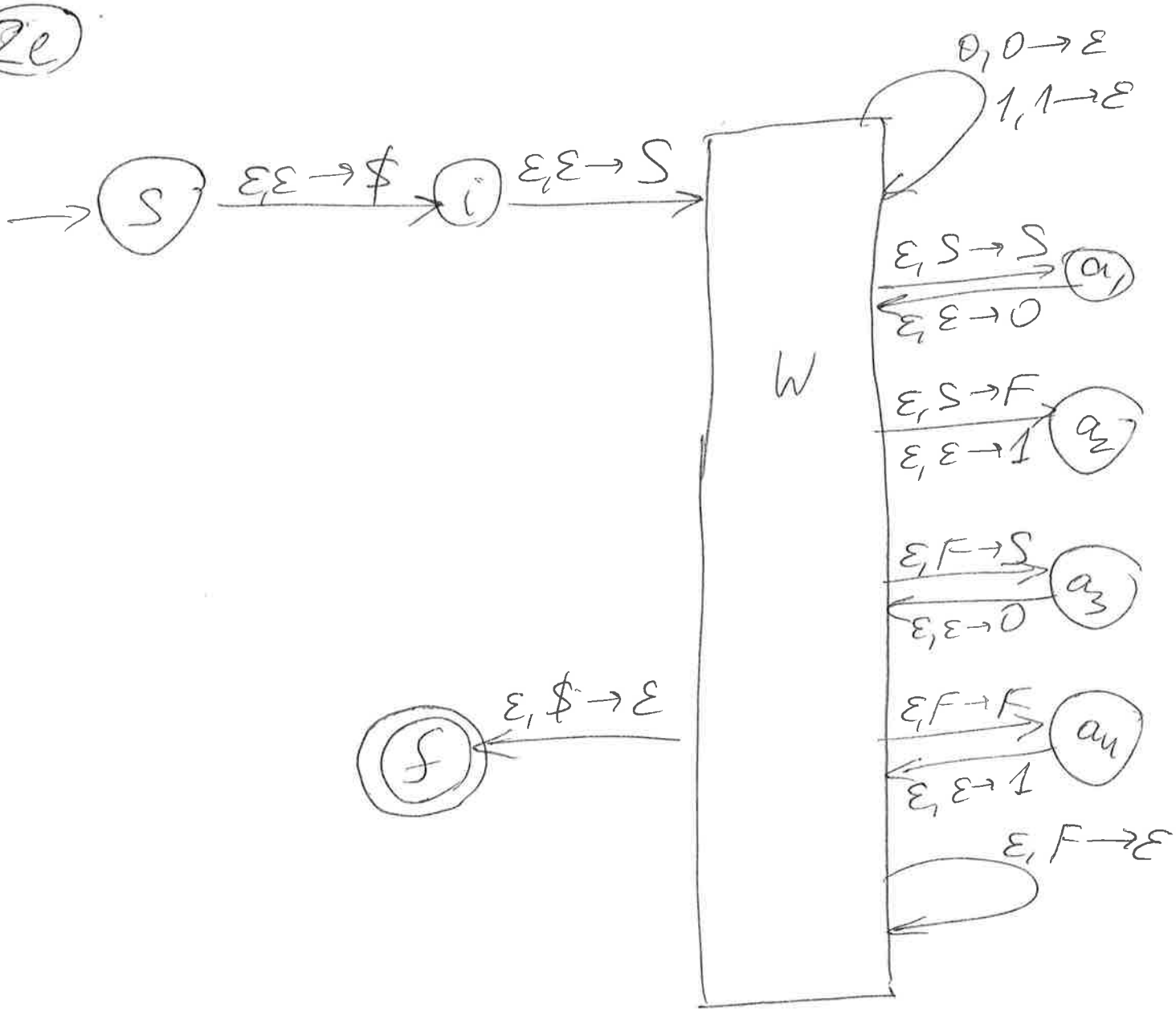
Prel. stage
~~$S_0 \to S$~~

Stage 0:
$S \to 1$
$F \to 1$

Stage 1:
~~$S_0 \to 0S$~~
~~$S_0 \to 1F$~~
$S_0 \to 1$

Stage 2
$V_0 \to 0$
$V_1 \to 1$
$S \to V_0 S$
$S \to V_1 F$
$F \to V_0 S$
$F \to V_1 F$
$S_0 \to V_0 S$
$S_0 \to V_1 F$

2e)



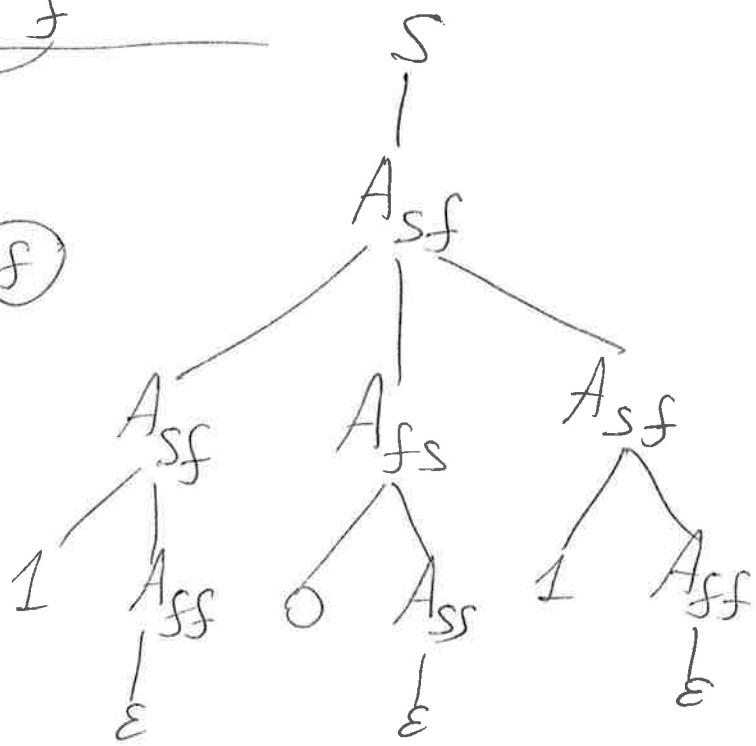
read				1			0			1			
state	s	i	w	a_2	w	w	a_3	w	w	a_2	w	w	w	f
stack	$	S	F	1	F	S	0	S	F	1	F	$		
		$	$	F	$	$	S	F	$	F	$			
				$			$			$				

(2f)

read	1	0	1
state	S	f	f
stack			

$A_{sf} \rightarrow 1 A_{ff} \varepsilon = 1 A_{ff}$

$A_{fs} \rightarrow 0 A_{ss} \varepsilon = 0 A_{ss}$

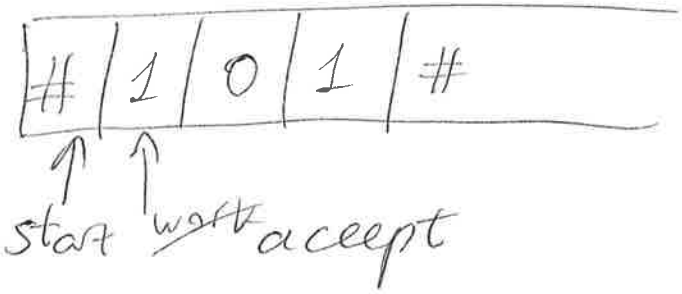


(3a)

start, # → work, R

work, 1 → accept

work, 0 → reject



3bc

start, # → s, R

s, 0 → s, R

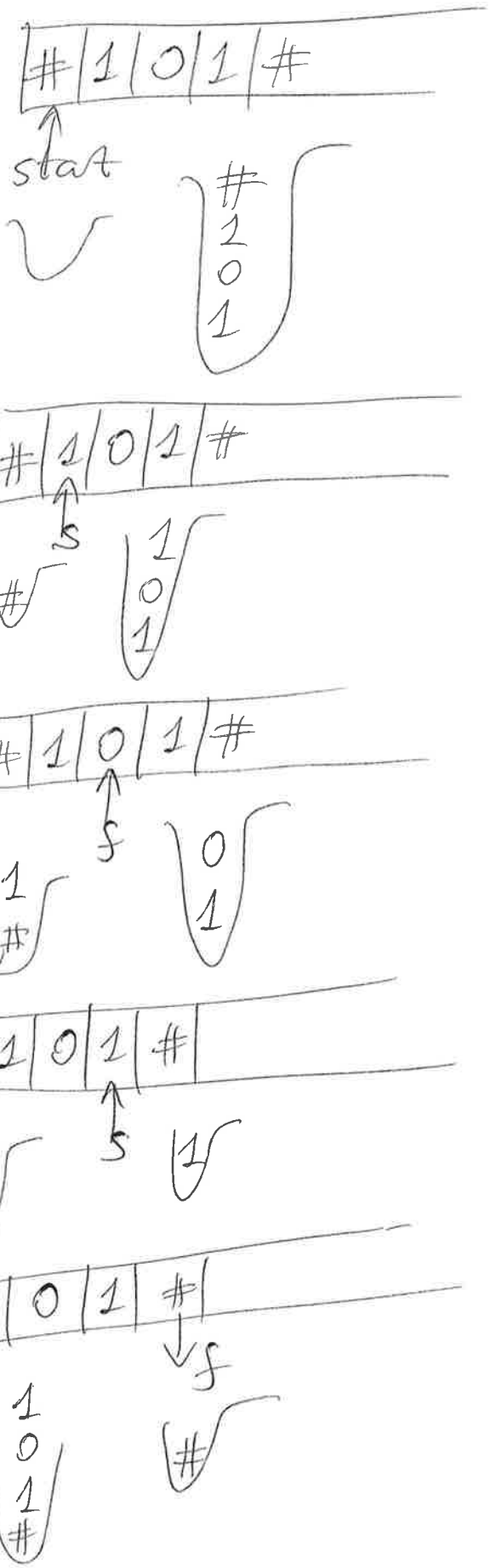
s, 1 → f, R

f, 0 → s, R

f, 1 → f, R

s, # → reject

f, # → accept



3de

Binary

start, # → skip, R

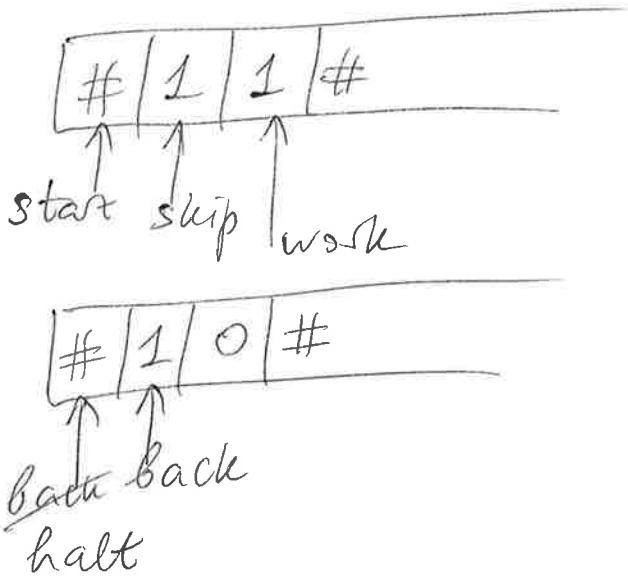
skip, $\begin{matrix}0\\1\end{matrix}$ → work, R

work, 0 → 1, R

work, 1 → 0, L, back

back, $\begin{matrix}0\\1\end{matrix}$ → L

back, # → halt



Unary

start, # → move, R

move, 1 → R

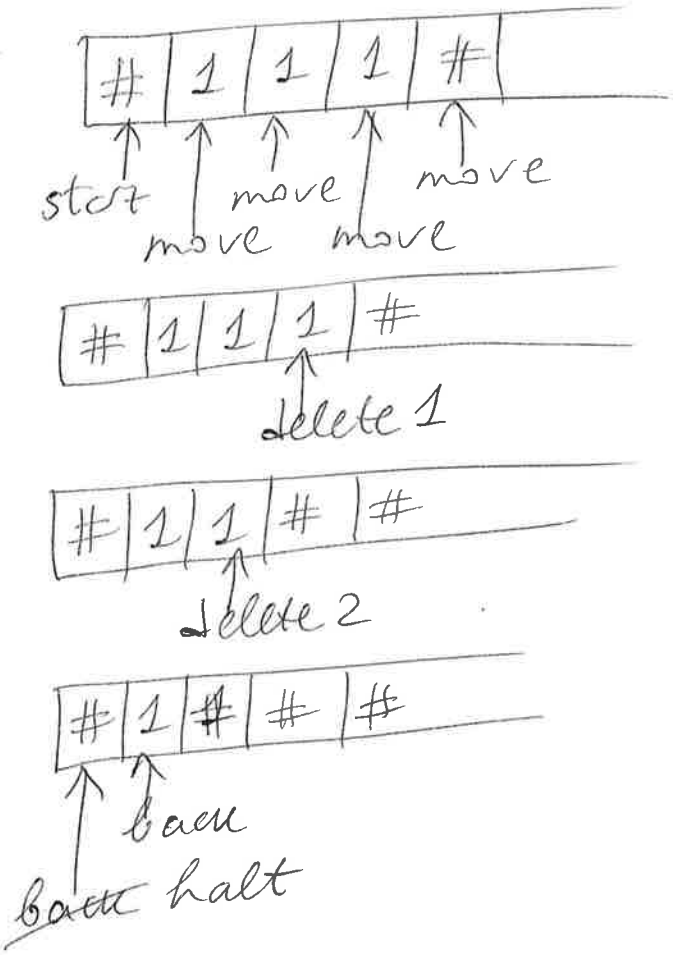
move, # → delete1, L

delete1, 1 → #, delete2, L

delete2, 1 → #, back, L

back, 1 → L

back, # → halt



4a. Church-Turing thesis states that any function that can be computed on any physical device can also be computed by a Turing machine (or, equivalently, by a Java program).

Whether this statement is true or not depends on the properties of the physical world. Thus, this statement is not a mathematical theorem, it is a statement about the physical world.

4b. The halting problem is the problem of checking whether a given program p halts on given data d. We can prove that it is not possible to have an algorithm haltChecker(p,d) that always solves this program by contradiction. Indeed, suppose that such an algorithm – i.e., such a Java program – exists. Then, we can build the following auxiliary Java program:

```
public static int aux(String x)
{if(haltChecker(x,x))
  (while(true) x= x;}
else{return 0;}}
```

If aux halts on aux, then haltChecker(aux,aux) is true, so the program aux goes into an infinite loop – and never halts. On the other hand, if aux does not halt on aux, then haltChecker(aux,aux) is false, so the program aux returns 0 – and thus, halts. In both cases, we get a contradiction, which proves that haltChecker is not possible.

4c. An algorithm A is called feasible if its running time $t_A(x)$ on each input x is bounded by some polynomial $P(\text{len}(x))$ of the length $\text{len}(x)$ of the input: $t_A(x) \leq P(\text{len}(x))$. In other words, the algorithm is feasible if for each length n, the worst-case complexity $t_A^w(n) = \max\{t_A(x) : \text{len}(x) = n\}$ is bounded by a polynomial: $t_A^w(n) \leq P(n)$.

An algorithm A is called practically feasible if for every input of reasonable length, it finishes its computations in reasonable time.

Time complexity $t_A^w(n) = 10^{200}$ is a constant – thus a polynomial, so from the viewpoint of the formal definition, it is feasible. However, this number is larger than the number of particles in the Universe, so it is clearly not practically feasible.

On the other hand, the function $\exp(10^{-200} \cdot n)$ is an exponential function and thus, grows faster than an polynomial, but even for largest realistic lengths n – e.g., for $n = 10^{18}$ – the resulting value is smaller than 3 and is, thus, perfectly practically feasible.

4d. P is the class of all the problems that can be solved in polynomial time.

NP is the class of all the problems for which, once we have a candidate for a solution, we can check, in polynomial time, whether it is indeed a solution.

A problem is called NP-hard if every problem from the class NP can be reduced to this problem.

A problem is called NP-complete if it is NP-hard and itself belongs to the class NP.

In general, optimization problems cannot be NP-hard, since they are usually not in the class NP: if someone proposes a candidate for a solution, how can we check that it is indeed a solution? We can compare the proposed solution with all other possible solutions, but there may be exponentially many of them, so this comparison cannot be done is feasible time.

At present, no one knows whether P is equal to NP. Most computer scientists believe that these two classes are different.

4e. When someone proves that a problem is NP-complete, the negative consequence is that, unless P = NP, no feasible algorithm is possible that would solve all instances of this problem.

The positive consequence is – since all other problems can be reduced to this problem – that whenever we have a good algorithm for solving some instances of this problem, we automatically get good algorithms for solving some instances of all other problems from the class NP.

An example of an NP-complete problem is propositional satisfiability:

- given: a propositional formula, i.e., any expression obtained from Boolean variables by using "and", "or", and "not",
- find: the values of the Boolean variables that makes this formula true.

4f. A language L is called *decidable* if there exists an algorithm (or, equivalently, a Turing machine) that:

- given a word,
- returns "yes" or "no" depending on whether this word belongs to this language or not.

An example is the language of all even numbers.

A language is called *semi-decidable*, *Turing-recognizable*, or *recursively enumerable* if there exists a Turing machine that:

- given a word w,
- returns "yes" if and only if the word w belongs to the language L.

An example of a semi-decidable language which is not decidable is the set of all the pairs (p, d) for which the program p halts on data d.