Aalto University
School of Science

CS-C2160 Theory of Computation

Lecture 2. Finite Automata

Pekka Orponen
Aalto University
Department of Computer Science

Spring 2022

Topics

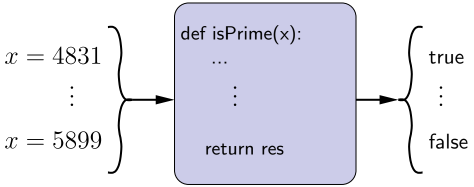
- State diagrams and transition tables
- Programming with finite automata
- Formal definition of finite automata
- * Excursion: Extensions of finite automata

Material:

- Sections 2.1–2.3 in Finnish lecture notes
  (or Section 1.1 in the Sipser book)

Recap: Languages and automata

- Computer programs and devices that solve computational problems can be seen as automata:



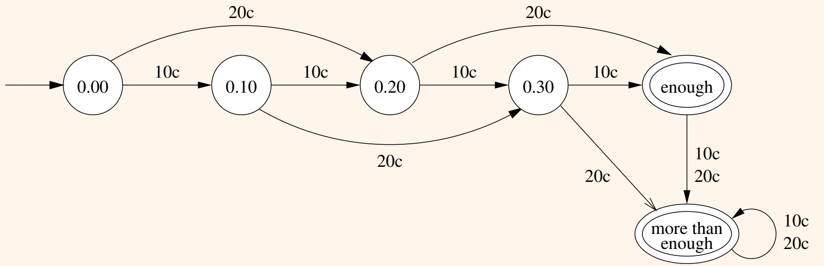
- *Alphabet*: a finite set of *symbols*, e.g., $\Sigma = \{0, 1, 2, \ldots, 9\}$.
- *Language*: a set of strings over Σ, e.g.,
  - $P = \{x \in \Sigma^\star \mid x \text{ is a prime}\} = \{2, 3, 5, 7, 11, 13, \ldots\}$
  - $F = \{z \in \{0x\texttt{00}, 0x\texttt{01}, \ldots, 0x\texttt{FF}\}^\star \mid$
    z is a virus-free machine-language program$\}$
- Given a problem, what kind of automaton is required to solve it (that is, to recognise a given language)?
- Can all problems be solved with some (effective) automaton?

Finite Automata (aka Finite State Machines)

2.1 State diagrams and transition tables

- We first study computing systems that have only finitely many possible states.
- Such systems can be modelled as *finite automata* (FA, also called finite state machines, FSM). (Wikipedia pages: FA, FSM.)
- There are many alternative representations for FA: state diagrams, transition tables, ...

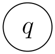
Example: A simple coffee machine



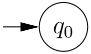
The state diagram above solves the decision problem "Has enough money been inserted for a cup of coffee?"

- In general, finite automata can be used to model procedures for solving simple decision (yes/no) problems.
- There are also variants of FA with more expressive output (e.g., Moore and Mealy machines) but these are not studied in detail on this course.

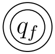
State diagram notation:



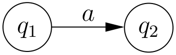
A state q



A start state q_0 (also called initial state)

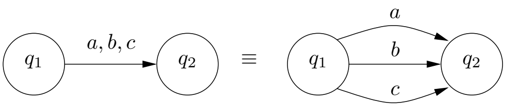


An accepting state q_f



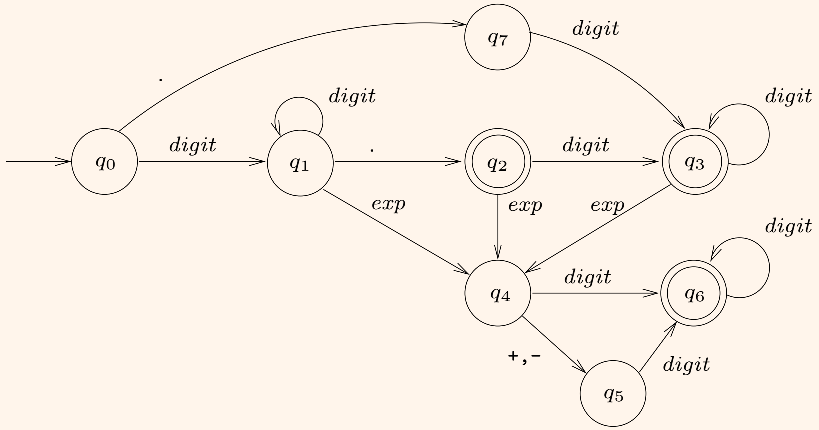
A transition from state q_1 to state q_2 on input symbol a

An abbreviation:



Example:

Floating point numerals in the C programming language.



Abbreviations: *digit* $= \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$, *exp* $= \{\texttt{E}, \texttt{e}\}$.

Representing an FA with a *transition table*: describe the successor state as a function of (i) the current state and (ii) the input symbol.

Example:

Transition table for the floating point automaton:

		$digit$	$.$	exp	$+$	$-$
$\rightarrow$	q_0	q_1	q_7			
	q_1	q_1	q_2	q_4		
$\leftarrow$	q_2	q_3		q_4		
$\leftarrow$	q_3	q_3		q_4		
	q_4	q_6			q_5	q_5
	q_5	q_6				
$\leftarrow$	q_6	q_6				
	q_7	q_3				

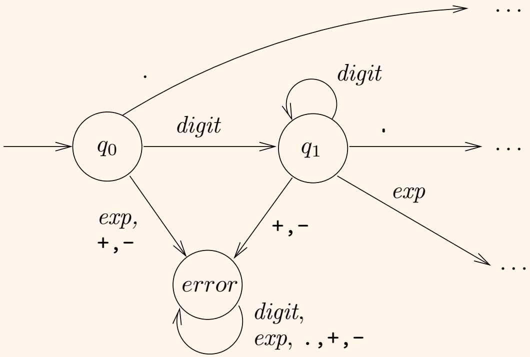
where "$\rightarrow$" marks the initial state and "$\leftarrow$" denotes an accept state.

Question: What are the empty cells in the transition table?

Answer: Empty cells in transition tables, and similarly missing edges in state diagrams, correspond to ”error situations” in the automaton.
If the automaton takes such a transition, the input is rejected.
Formally, the automaton contains a specific error state but we omit drawing the state and the transitions to it for the sake of clarity.

Example:

The complete state diagram of the floating point automaton would look like this:



and the complete transition table would be:

		$digit$	$.$	exp	$+$	$-$
$\rightarrow$	q_0	q_1	q_7	$error$	$error$	$error$
	q_1	q_1	q_2	q_4	$error$	$error$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\leftarrow$	q_6	q_6	$error$	$error$	$error$	$error$
	$error$	$error$	$error$	$error$	$error$	$error$

2.2 Programming with finite automata

- Given a finite automaton, it is straightforward to implement it as a corresponding program in some programming language.

Example:

A Python program based on the floating point automaton, testing whether a string is a valid floating point numeral in C.

☞CODE

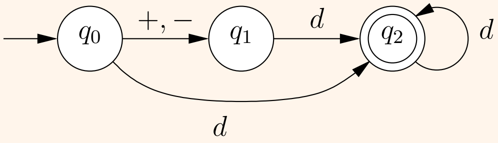
```
from sys import stdin
q=0
for c in stdin.readline().strip("\n"):
    if q==0:
        if c.isdigit(): q=1
        elif c==".": q=7
        else: q=99
    elif q==1:
        if c.isdigit(): q=1
        elif c==".": q=2
        elif c=="E" or c=="e": q=4
        else: q=99
...
    elif q==7:
        if c.isdigit(): q=3
        else: q=99

if q in [2,3,6]: print("Is a valid floating point numeral")
else: print("Not a floating point numeral")
```

Finite automata with semantic actions

- One can add actions to the states and transitions of (a program based on) a finite automaton.

Example:

- An automaton recognising octal numerals:



$$\begin{tikzcd} \arrow[r] & q_0 \arrow[r, "{+,-}"] \arrow[rr, bend right, "d"'] & q_1 \arrow[r, "d"] & q_2 \arrow[loop right, "d"] \end{tikzcd}$$

Abbreviation: $d = \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{7}\}$.

- Let us build a program from this that computes the number value of a given octal numeral, and outputs it in decimal notation.

A corresponding Python program for the syntax check:

☞CODE

```
from sys import stdin

q=0
for c in stdin.readline().strip("\n"):
    if q==0:
        if c=="+" or c=="-": q=1
        elif c in "01234567": q=2
        else: q=99
    elif q==1:
        if c in "01234567": q=2
        else: q=99
    elif q==2:
        if c in "01234567": q=2
        else: q=99
if q==2: print("Octal numeral")
else: print("Not an octal numeral")
```

Adding operations that compute the value of the input octal numeral and output it in decimal notation:

☞CODE

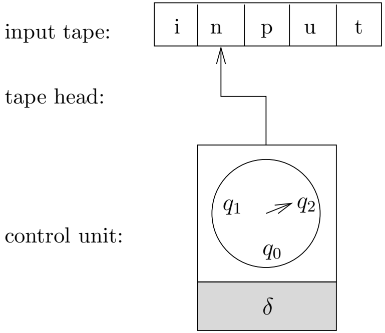
```
from sys import stdin

q=0
sgn=1   # SEM: sign
val=0   # SEM: absolute value
for c in stdin.readline().strip("\n"):
    if q==0:
        if c=="+": q=1
        elif c=="-": sgn=-1; q=1
        elif c in "01234567": val=int(c); q=2
        else: q=99
    elif q==1:
        if c in "01234567": val=int(c); q=2
        else: q=99
    elif q==2:
        if c in "01234567": val=8*val+int(c); q=2
        else: q=99
if q==2: print("Octal numeral; decimal presentation is", sgn*val)
else: print("Not an octal numeral")
```

* Other application examples

- Finite automata can used when designing and implementing behaviours of actors in computer games, see e.g. https://www.techopedia.com/finite-state-machine-how-it-has-affected-your-gaming-for-over-40-years/2/33996 and https://www.youtube.com/watch?v=JyF0oyarz4U/.
- The graphical state diagram editor used in the course's computerised home assignments has itself been designed as an automaton with transitions of the form: "If the present state is 'adding transitions' and an event 'mouseclick on an edge' is received, then go to state 'modifying edge symbol set' ", etc.
- Also see Wikipedia for automata-based programming and event-driven finite state machines.

2.3 Formal definition of a finite automaton

Mechanistic model:



A finite automaton M consists of

- a *control unit* with a finite number of states and a *transition function* δ, and
- an *input tape* that is accessed symbol-by-symbol with a *tape head*.

The behaviour of the automaton on an input string is as follows:

- In the beginning, the automaton is in its *start state* q_0, the input is written on the input tape, and the tape head is pointing to the first symbol of the input.
- In one step, the automaton:
  - reads the symbol under the tape head,
  - decides, based on its current state and the symbol, the next state,
  - changes to the next state, and
  - moves the tape head to the next symbol on the input tape.
- The automaton halts after the last symbol on the input tape has been processed.
- If the current state at the time of halting is a special *accept state*, the automaton *accepts* the input. Otherwise, it *rejects* the input.
- The language *recognised* by the automaton is the set of strings it accepts.

The verbal description above may have many interpretations, e.g., what happens when the empty string ε is input?

A precise mathematical formulation:

Definition 2.1 (Finite automata)

A *finite automaton* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F),$$

where

- Q is a finite set of *states*,
- Σ is a finite set called *alphabet*,
- $\delta : Q \times \Sigma \to Q$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *final* or *accepting states*.

Example:

Formal presentation of the floating point automaton:

$$\begin{aligned} M &= (\{q_0, \ldots, q_7, \textit{error}\}, \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{.}, \texttt{E}, \texttt{e}, \texttt{+}, \texttt{-}\}, \\ &\quad \delta, q_0, \{q_2, q_3, q_6\}), \end{aligned}$$

where δ is as described in the complete transition table earlier:

$$\delta(q_0, \texttt{0}) = \delta(q_0, \texttt{1}) = \cdots = \delta(q_0, \texttt{9}) = q_1,$$

$$\delta(q_0, \texttt{.}) = q_7, \quad \delta(q_0, \texttt{E}) = \delta(q_0, \texttt{e}) = \textit{error},$$

$$\delta(q_1, \texttt{.}) = q_2, \quad \delta(q_1, \texttt{E}) = \delta(q_1, \texttt{e}) = q_4,$$

etc.

Formal definition for the semantics of FA

The previous definition describes the *structure* of a finite automaton but not yet how it works. The behaviour (i.e., semantics) of an automaton $(Q, \Sigma, \delta, q_0, F)$ can also be described mathematically with a few additional definitions.

- A *configuration* of the automaton is a pair $(q, w) \in Q \times \Sigma^\star$.
- The *start configuration* on input $x \in \Sigma^\star$ is the pair (q_0, x).
- *Intuition:* q is the current state of the automaton and w is the part of the input that has not yet been processed.

Example

The following (among many others) are configurations of the floating point automaton shown earlier: $(q_0, \texttt{0.25E2})$, $(q_0, \texttt{EE..33})$, $(\textit{error}, \texttt{E..33})$, $(q_1, \texttt{.25E2})$, (q_6, ε).

On input `.242E10`, the start configuration is $(q_0, \texttt{.242E10})$.

- A configuration (q,w) *leads in one step* or *leads directly* to configuration (q',w'), denoted by

$$(q,w) \underset{M}{\vdash} (q',w'),$$

if $w = aw'$ ($a \in \Sigma$) and $q' = \delta(q,a)$. We say configuration (q',w') is the *immediate successor* of configuration (q,w).
- *Intuition:* if the automaton is in state q and reads the first symbol a of the yet unprocessed string $w = aw'$ on the tape, it will move to state q' and move the tape head one step right, so that the remaining string on the tape is w'.
- If the automaton M is clear from the context, we simply write

$$(q,w) \vdash (q',w').$$

Example:

Consider again the floating point automaton. Why does

- $(q_0, \texttt{0.25E2}) \vdash (q_1, \texttt{.25E2})$ hold?
- $(q_0, \texttt{0.25E2}) \vdash (q_6, \texttt{5E2})$ not hold?
- configuration (q_6, ε) not have immediate successors?

A

- A configuration (q, w) *leads* to a configuration (q', w'), i.e., the configuration (q', w') is a *successor* of (q, w), denoted by

$$(q, w) \underset{M}{\vdash}^{*} (q', w'),$$

if there exists a sequence $(q_0, w_0), (q_1, w_1), \ldots, (q_n, w_n)$ of configurations, for some $n \geq 0$, such that

$$(q, w) = (q_0, w_0), \quad (q_0, w_0) \underset{M}{\vdash} (q_1, w_1), \quad (q_1, w_1) \underset{M}{\vdash} (q_2, w_2)$$
$$\ldots \quad (q_{n-1}, w_{n-1}) \underset{M}{\vdash} (q_n, w_n) \quad \text{and} \quad (q_n, w_n) = (q', w').$$

- As a special case when $n = 0$, we have $(q, w) \underset{M}{\vdash}^{*} (q, w)$ for any configuration (q, w).
- Again, when the automaton M is clear from the context, we simply write

$$(q, w) \vdash^{*} (q', w').$$

- The automaton M *accepts* a string $x \in \Sigma^\star$ if

$$(q_0, x) \underset{M}{\vdash}^* (q_f, \varepsilon) \qquad \text{for some } q_f \in F;$$

otherwise M *rejects* x.

- In other words, the automaton accepts x if its start configuration on x leads to a configuration with an accept state and no remaining input.
- The *language recognised by the automaton M* is defined as

$$\mathcal{L}(M) = \{x \in \Sigma^\star \mid (q_0, x) \underset{M}{\vdash}^* (q_f, \varepsilon) \text{ for some } q_f \in F\}.$$

Example:

The behaviour of the floating point automaton on the input “`0.25E2`”:

$$\begin{array}{llllll} (q_0, \texttt{0.25E2}) & \vdash & (q_1, \texttt{.25E2}) & \vdash & (q_2, \texttt{25E2}) \\ & \vdash & (q_3, \texttt{5E2}) & \vdash & (q_3, \texttt{E2}) \\ & \vdash & (q_4, \texttt{2}) & \vdash & (q_6, \varepsilon). \end{array}$$

Because $q_6 \in F = \{q_2, q_3, q_6\}$, we have $\texttt{0.25E2} \in \mathcal{L}(M)$.

* An alternative definition (from the Sipser book)

- Acceptance can also be defined without using “configurations”.
- In this case, the remaining input string is not directly visible in the presentation.

Definition

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA.
- M *accepts* a string $w = w_1 w_2 \ldots w_n \in \Sigma^\star$ if there exists a finite sequence $r_0 r_1 r_2 \ldots r_n \in Q^\star$ of states such that
  - $r_0 = q_0$
  - $\delta(r_i, w_{i+1}) = r_{i+1}$ for all $0 \leq i < n$
  - $r_n \in F$
- $\mathcal{L}(M) = \{w \in \Sigma^\star \mid M \text{ accepts } w\}$.

Example:

The floating point automaton accepts the string “`0.25E2`” as the state sequence $q_0 q_1 q_2 q_3 q_3 q_4 q_6$ fulfills all the requirements above.

More examples

Example:

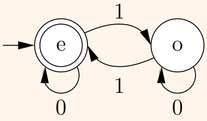
The languages

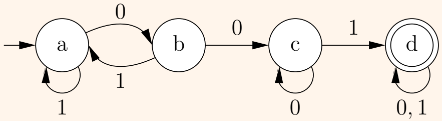
$$\{w \in \{0,1\}^\star \mid w \text{ contains an even number of symbol } 1\}$$

and

$$\{w \in \{0,1\}^\star \mid w \text{ contains } 001 \text{ as a substring}\}$$

can be recognised with the following automata:





An "intuitive interpretation" for the state *a* could be "we have not yet seen the substring 001 and the previous symbol was 1". What could be the "explanations" for the other states in the automaton?

Lemma 2.1

If a language $L \subseteq \Sigma^\star$ can be recognised with a finite automaton, then the complement language $\overline{L} = \{w \in \Sigma^\star \mid w \notin L\}$ can be recognised with a finite automaton as well.

Proof

Let $M_L = (Q, \Sigma, \delta, q_0, F)$ be any FA that recognises the language L (i.e., has $\mathcal{L}(M_L) = L$).
As the state of M_L at the end of its computation on any input string is unique and well-defined, we get an automaton for the complement language $\overline{L} = \{w \in \Sigma^\star \mid w \notin L\}$ simply by replacing all the accept states in M_L by non-accept ones and vice versa. (Note that the presentation of M_L is here assumed to be complete, i.e. all the entries in the transition table/diagram have been filled in.)
Thus, the FA $M_{\overline{L}} = (Q, \Sigma, \delta, q_0, Q \setminus F)$ recognises $\overline{L}$.

Example:

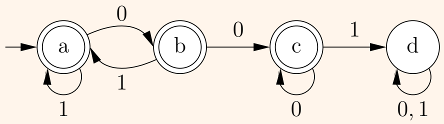
By applying the construction of the above proof to the automaton of the previous example recognising the language

$$L = \{w \in \{0,1\}^\star \mid w \text{ contains } 001 \text{ as a substring}\},$$

we get an automaton recognising the complement language

$$\overline{L} = \{w \in \{0,1\}^\star \mid w \text{ does not contain } 001 \text{ as a substring}\}.$$

The automaton is



Lemma 2.2

If languages $A, B \subseteq \Sigma^\star$ can be recognised with finite automata, then so can the language $A \cap B = \{w \in \Sigma^\star \mid w \in A \text{ and } w \in B\}$.

Proof

Proof by construction.

Let $M_A = (Q_A, \Sigma, \delta_A, q_{A,0}, F_A)$ and $M_B = (Q_B, \Sigma, \delta_B, q_{B,0}, F_B)$ be some FA that recognise the languages A and B, respectively.

We build an automaton $M_{A\cap B} = (Q_{A\cap B}, \Sigma, \delta_{A\cap B}, q_{A\cap B,0}, F_{A\cap B})$ that "simulates" both automata at the same time and accepts a string if and only if both automata would.

$M_{A\cap B}$ is called the *(synchronous) product* automaton, and defined as:

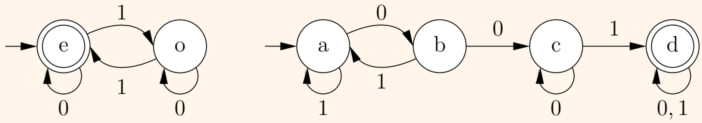
- $Q_{A\cap B} = Q_A \times Q_B$, meaning that the states are pairs that record in which states the simulated automata A and B would be.
- $q_{A\cap B,0} = (q_{A,0}, q_{B,0})$, indicating that in the beginning both simulated automata are in their start states.
- $\delta_{A\cap B}((q_a, q_b), \sigma) = (\delta_A(q_a, \sigma), \delta_B(q_b, \sigma))$, meaning that $M_{A\cap B}$ moves to a new state that corresponds to the states of A and B when they would read the same symbol σ.
- $F_{A\cap B} = F_A \times F_B$, so that $M_{A\cap B}$ accepts exactly when both of the simulated automata would.

Now one could prove, by using induction on the length of the input string, that if, after reading the input string, automaton A is in state q_a and B is in state q_b, then automaton $M_{A\cap B}$ is in state (q_a, q_b).

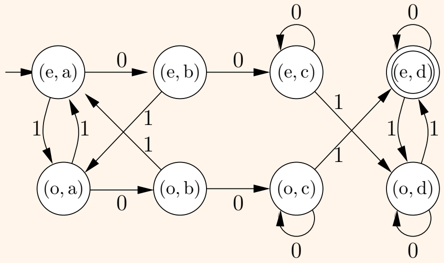
Example:

By using our earlier examples and the construction in the previous proof, we can build an automaton that accepts exactly the strings that contain an even number of symbol 1 *and* 001 as a substring.

Component automata:



The product automaton:



* Excursion: Extensions of Finite Automata

* Extensions of finite automata

Finite automata are perhaps the simplest automata class. There are many extensions, those described shortly in the next slides as well as:

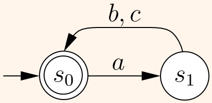
- Pushdown automata, where the input tape is replaced by a read-write stack. These are discussed later in the course.
- Timed automata that extend FA with real-valued clocks (see, e.g., this tutorial article and the Uppaal tool).
- Hybrid automata that allow more general use of real-valued variables. They are used to model and analyse systems that control and interact with physical processes.
- Turing machines, where the input tape is replaced by an infinite (more precisely, infinitely extendible) read-write storage tape. These are also discussed later in the course.

* Automata accepting infinite strings

- In this course, we only consider finite input strings.
- However, in some applications infinite strings are needed: for instance, the executions of many reactive systems (servers, protocols, etc) are not expected to terminate but to continue forever.
- Many automata classes have been defined to handle infinite strings.
- E.g., Büchi automata accept an infinite string if an accept state is visited infinitely often.
- One application example: the Spin verification tool.

Example:

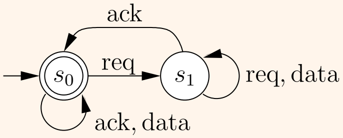
A Büchi automaton accepting the infinite strings over $\{a, b, c\}$ whose even indices have the symbol a.



Now $abacacaba\ldots$ is accepted but $bbacacaba\ldots$ and $abacaccba\ldots$ are rejected.

Example:

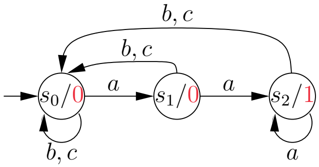
A Büchi automaton that accepts the infinite strings over $\{\mathrm{req}, \mathrm{ack}, \mathrm{data}\}$ in which each req symbol (“request”) is followed at some point by an ack symbol (“acknowledgement”).



* Transducers

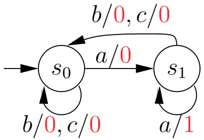
Finite state transducers do not accept/reject input but transform it into output.

- Moore machines associate output to states:



input:	a	b	c	a	a	a	c	...	
state:	s_0	s_1	s_0	s_0	s_1	s_2	s_2	s_0	...
output:		0	0	0	0	1	1	0	...

- Mealy machines associate output to state transitions, which allows it to also depend on the current input symbol:



input:	a	b	c	a	a	a	c	...	
state:	s_0	s_1	s_0	s_0	s_1	s_1	s_1	s_0	...
output:	0	0	0	0	1	1	0	...	