A Subset of the URM Language; FA and NFA

This Note turns to a special case of the URM programming language that we call *Finite Automata*, for short *FA*.

This part presents almost a balance of How To and Limitations of Computing topics.

Main feature of the latter will be the so-called "*Pumping Lemma*".

0.1. The FA

The FA (*programming language*)[†] is introduced informally as a modified and restricted URM.

This new URM model will have explicit "**read**" instructions.[*]

Secondly, *any specific URM under this model* ***will ONLY have ONE variable*** that we may call generically "**x**".

This variable will always be of *type* *single-digit*; it cannot hold arbitrary integers, rather it can only hold *single digits* as *values*.

[†]Note that some texts look at it as a "machine", hence the terminology "automaton".

[*]In Notes #2 we explained why explicit *read* instructions are theoretically as redundant as explicit *write* instructions are.

The FA has no instructions —other than "**read**"— compared to the FULL URM, except for a simplified **if-goto** instruction.

In the absence of a **stop** instruction, how does a computation halt?

We postulate that our modified URMs halt simply by reading something *that does not belong*, that is, it saw in the input stream an object that is *not* a member of the *input alphabet* of permissible digits.

Such an "illegal" symbol serves as an end-marker of the useful stream digits that constitute the *input string* over the given alphabet. As such it is often called an "*end-of-file*" marker, for short, *eof*.

This *eof*-marker is any "illegal" symbol, that is, a symbol not in the *particular FA's INPUT ALPHABET*.

Thus the modified URM halts if ***IFF*** *it runs out of input, as this is signaled by it reading something* NOT in its input alphabet*.*

Our insistence on a URM-like model for the automaton *will be confined in this brief motivational introduction* and is only meant to illustrate the indebtedness of the finite automata model to the general URM model of Notes #2, as promised above.

The FA has, for *each* label L, a **group** of instructions as follows.

The typical group-instruction of an automaton.

$$L: \begin{cases} \textbf{read} \\ \textbf{if } \mathbf{x} = a \textbf{ then goto } M' \\ \textbf{if } \mathbf{x} = a' \textbf{ then goto } M'' \\ \vdots \\ \textbf{if } \mathbf{x} = a^{(n)} \textbf{ then goto } M^{(n)} \\ \textbf{if } \mathbf{x} = eof \textbf{ then halt} \end{cases}$$

where L and $M', \ldots, M^{(n)}$ are labels —*not necessarily distinct*— and $a, a', \ldots, a^{(n)}$ are **all** the possible digit values in the context of a specific URM program, that is, $\{a, a', \ldots, a^{(n)}\}$ *is* the *input alphabet*.

The empty string, λ, will never be part of a FA's input alphabet.

For any particular *FA (program)* —a particular FA, as we say (omitting "program")— labels, in practice, *are not restricted to be numerical* nor even to be consecutive (if numerical).

▶ However, *one* instruction's placement is significant.

It is often identified by a label such as "0", or "q_0", or some such symbol and **is placed at the very beginning of the program**.

This instruction's label is called the **initial state** of the specific automaton. Indeed, all labels in an automaton are called states in the literature.

Pause. A finite automaton does not care about the order of its other instructions, since they will be reachable by the goto-structure as needed wherever they are.◀

The semantics of the "typical" instruction above is:

- Read into the variable **x** the first unread digit-value from some "external (to the FA) input stream" that is waiting to be read.
- Then move to the *next instruction as is determined* by the $a^{(i)}$s (or the *eof*) in the if-cases above (p.5).

In order to have the FA make a decision about the input string it just read, we (this is part of the design of the particular FA program) partition the instruction-labels of any given FA into two types: **accepting** and **rejecting**.

Their role is as follows: Such an FA, when it has halted,

Pause. *When* or *if*?◀

will have finished scanning a sequence of digits —a string over its alphabet.

This string is *accepted* if the program halted while in an *accepting state*, otherwise the input is *rejected*.

0.1.1 Definition. (The Language of an FA)
The language decided by a FA M is called in the literature "the Language **accepted** by M". It is, of course,

$$L(M) \stackrel{Def}{=} \{x : x \text{ is accepted by automaton } M\}$$

□

Since an FA cannot "write", i.e., cannot change the contents of $\mathbf{x}$ —since it does not have any of the instructions $\mathbf{x} \leftarrow c$, $\mathbf{x} \leftarrow \mathbf{x}+1$, $\mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$— we need the *type* of state to "code" the yes/no (accept/reject) answer.

0.2. Deterministic Finite Automata and their Languages

0.2.1 Example. Consider the FA below that operates over the input alphabet $\{0, 1\}$

$$0 : \begin{cases} \textbf{read} \\ \textbf{if x} = 0 \textbf{ then goto } 0 \\ \textbf{if x} = 1 \textbf{ then goto } 1 \\ \textbf{if x} = \textit{eof} \textbf{ then halt} \end{cases}$$

$$1 : \begin{cases} \textbf{read} \\ \textbf{if x} = 0 \textbf{ then goto } 1 \\ \textbf{if x} = 1 \textbf{ then goto } 0 \\ \textbf{if x} = \textit{eof} \textbf{ then halt} \end{cases}$$

What does this program do? Once we have the graph model, we will elaborate on what the above automaton actually does. LATER!

In particular we will look into two cases:

- When only state 0 is accepting.
- When only state 1 is accepting.

$\square$

0.2.1. FA as Flow-Diagrams

Moving away from the URM-like programming language for automata, we next consider a "flow chart" or "flow diagram" formalisation. This is achieved by first abstracting an instruction

$$L: \ \textbf{read; if x} = a \textbf{ then goto } M \tag{1}$$

as the configuration below:

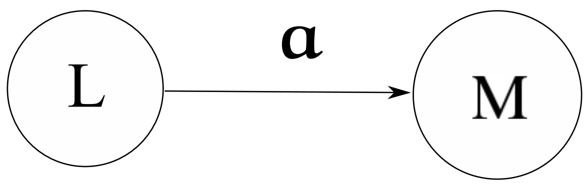
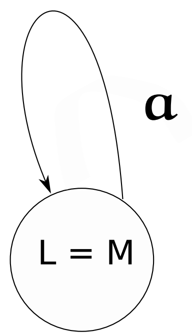


Figure capturing (1) above

Thus the "read" part is implicit, while the labeled arrow that connects the states L and M denotes exactly the semantics of (1).

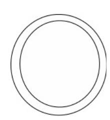
Therefore, an entire automaton can be viewed as a *directed graph* —that is, a finite set of (possibly) labeled circles, the *states*, and a finite set of arrows, the *transitions*, the latter labeled by members of the automaton's input alphabet.

An arrow label a in the figure above represents "**if** $\mathbf{x} = a$ **then goto** M". The arrows or *edges* interconnect the states. If $L = M$, then we have the configuration

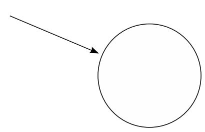


where the optional label could be L, or M, or $L = M$ (as above), or nothing.

We depict the partition of states into *accepting* and *rejecting* by using two concentric circles for each accepting state as below.



The special start state is denoted by drawing an arrow, that comes from nowhere, pointing to the state.



To summarise and firm up:

0.2.2 Definition. (FA as Flow Diagrams) A *finite automaton*, for short, *FA*, over the FINITE *input alphabet* Σ is a finite directed graph of circular nodes —the *states*— and interconnecting edges —the *transitions*— the latter labeled by members of Σ.

We impose a restriction to the automaton's structure:

▶ For every state L and every $a \in \Sigma$, there will be *precisely* one *edge*, labeled a, leaving L and pointing to some state M (possibly, $L = M$).

We say the automaton is *fully specified* (corresponding to the italics in the part "For every state L and every $a \in \Sigma$, *there will be* ...") and *deterministic* (corresponding to the italics in the part "there will be *precisely one* edge, ...").

This graph depiction of a FA is called its *flow diagram* and is akin to a programming "*flow chart*". □

0.2.3 Remark. (1) Thus, full specification makes the *transition function total* —that is, for any state-input pair (L, a) as argument, it will yield some state as "*output*".

On the other hand, *determinism* ensures that the transition function is indeed a *function* (single-valued).

(2) **On Digits.** Each "legal" input symbol is a member of the alphabet Σ, and vice versa. In the preamble of this chapter we referred to such legal symbols as "digits" in the interest of preserving the *inheritance* from the URM of Notes #2, the latter being a number-theoretic programming language.

But what *is* a "digit"? In binary notation it is one of 0 or 1. In decimal notation we have the digits $0, 1, \ldots, 9$. In *hexadecimal* notation† we add the "digits" a, b, c, d, e, f that have "values", in that order, $10, 11, 12, 13, 14, 15$. The objective is to have single-symbol, *atomic*, digits to avoid ambiguities in string notation.

Thus, a "digit" is an atomic symbol (unlike "10" or "11").

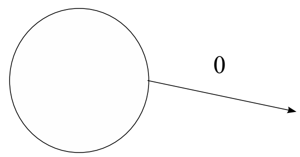
We will drop the terminology "digit" from now on.

Thus our automata alphabets are *finite sets of symbols* —any length-ONE symbols, period. □

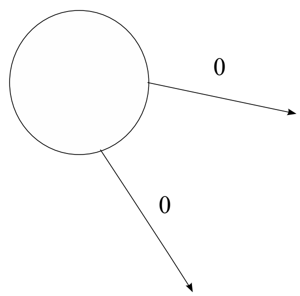
†Base 16 notation.

0.2.4 Example. Thus, if our alphabet is $A = \{0, 1\}$, then we cannot have the following configurations be part of a FA.

Nontotal Transition Function

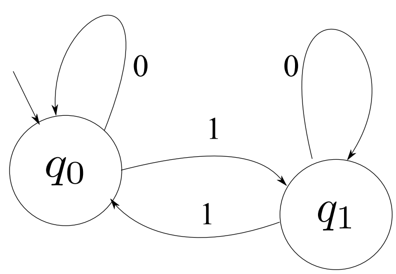


Non-determinism



$\square$

0.2.5 Example. The FA of the example of 0.2.1, in flow diagram form but with no decision on which state(s) is/are accepting is given below:



We wrote q_0 and q_1 for the states "0" and "1" of 0.2.1.

$\square$

Another way to define a FA without the help of flow diagrams is as follows:

0.2.6 Alternative Definition. (FA —Algebraically) A *finite automaton*, *FA*, is a toolbox $M = (Q, A, q_0, \delta, F)$,[‡] where

(1) Q is a finite set of states.

(2) A is a finite set of symbols; the *input alphabet*.

(3) $q_0 \in Q$ is the distinguished *start state*.

(4) $\delta : Q \times A \to Q$ is a *total function*, called the *transition function*.

(5) $F \subseteq Q$ is the set of accepting states; $Q - F$ is the set of rejecting states. $\square$

[‡] "M" is generic; for "machine".

0.2.7 Remark. Let us compare Definitions 0.2.2 and 0.2.6.

(1) The set of states corresponds with the nodes of the graph (flow diagram) model. It is convenient —but not theoretically necessary in general— to actually *name* (label) the nodes with names from Q.

(2) The A in the flow diagram model is not announced separately, but can be extracted as the set of all edge labels.

(3) q_0 —the start state by any name; q_0 being generic— in the graph model is recognised/indicated as the node pointed at by an arrow that emanates from no node.

(4) $\delta : Q \times A \to Q$ in the graph model is given by the arrow structure: Referring to the figure at the beginning of 0.2.1, we have $\delta(L, a) = M$. □

How does a FA compute? From the URM analogy, *we understand the computation of a FA consisting of successive*

- read moves
- attendant changes of state
- until the program halts (by reading the *eof*).
- At that point we proclaim that the string formed by the stream of symbols read is *accepted* or *rejected* according as the halted machine is in an accepting or rejecting state.

To formalise/mathematise FA computations as described above, we use snapshots or *Instantaneous Descriptions* (of a computation), for short *ID*s.

The IDs of the FA are very simple, since the machine (program) is incapable of altering the input stream.

You do not need to keep track of how the contents of variables change.

0.2.8 Remark. We recall from discrete mathematics, that a *binary relation* R is a set of *ordered pairs* and we prefer to write aRb instead of $(a,b) \in R$ or $R(a,b)$. For example, we write $a \le b$ if R is $\le$.

We also recall that the so-called *transitive closure* of a relation R, denoted R^+, is defined by

$aR^+b \stackrel{Def}{\equiv} aRa_1Ra_2 \ldots a_{m-1}Rb$, for some $a_i, i = 1, \ldots, m-1$

We note that

for all i, $a_iRa_{i+1}Ra_{i+2}$ is short for a_iRa_{i+1} **and** $a_{i+1}Ra_{i+2}$

just as $a \le b \le c$ means $a \le b$ **and** $b \le c$.

The *reflexive transitive closure* of R is denoted by R^* and is defined by

$$aR^*b \stackrel{Def}{\equiv} a = b \vee aR^+b$$

The following also are useful:

$$aR^mb \stackrel{Def}{\equiv} aRa_1Ra_2Ra_3Ra_4 \ldots a_{m-2}Ra_{m-1}Rb$$

that is, exactly m copies of R occur in the **R-chain** —or just "chain" if R is understood—

$$aRa_1Ra_2Ra_3Ra_4 \ldots a_{m-2}Ra_{m-1}Rb$$

Finally, "$aR^{<m}b$" means "aR^nb **and** $n < m$". □

0.2.9 Definition. (FA Computations; Acceptance) Let $M = (Q, A, q_0, \delta, F)$ be a FA, and x be an input string —that is, a string over A that is presented as a *stream of (atomic) input symbols from* A.

An *M-ID* or simply *ID* related to x is a string of the form tqu, where $q \in Q$, and $x = tu$.

Intuitively, the expression tqu means that the computing agent, the FA, is in state q and that the next input to process is the *first symbol* of u.

$$\overbrace{t}^{\text{processed}} \quad q \quad \underbrace{u}_{\text{to be processed}}$$

If $u = \lambda$ —and hence the ID is simplified to tq— then M has halted (has read *eof*; no more input).

Formally, an ID of the form tq has *no next ID*. We call it a *terminal ID*.

However, an ID of form $tqau'$, where $a \in A$, has a *unique* next ID; this one: $ta\widetilde{q}u'$, *just in case* $\delta(q, a) = \widetilde{q}$.

We write

$$tqau' \vdash_M ta\widetilde{q}u'$$

or, simply (if M is understood)

$$tqau' \vdash ta\widetilde{q}u'$$

and pronounce it "(ID) $tqau'$ *yields* (ID) $ta\widetilde{q}u'$".

We say that M *accepts the string* x iff, for some $q \in F$, we have $q_0x \vdash_M^* xq$.

The *language accepted by the FA* M is denoted generically by $L(M)$ and is the subset of A^* —this is notation for the set of **all** strings over the alphabet A[§]— given by $L(M) = \{x : (\exists q \in F)q_0x \vdash_M^* xq\}$.

An ID of the form q_0x is called a *start-ID*. □

0.2.10 Remark.

(I) Of course, $\vdash_M^*$ is the *reflexive transitive closure* of $\vdash_M$ and therefore $I \vdash_M^* J$ —where I (**not** necessarily a start-ID) and J (**not** necessarily terminal) are IDs— means that $I = J$ **or**, for some IDs I_m, $m = 1, \ldots, n-1$, we have an $\vdash_M$-chain

$$I \vdash_M I_1 \vdash_M I_2 \vdash_M I_3 \vdash_M \ldots \vdash_M I_{n-1} \vdash_M J \quad (1)$$

We say that we have an M-computation from I to J iff we have $I \vdash_M^* J$. We say simply *computation* if the "M-" part is understood.

[§] A^+, by definition, is $A^* - \{\lambda\}$.

(II) There is a tight relationship between computations and paths in a FA depicted as a graph.

To see this let us look at (1) above closer, namely, let $I = tp_1a_1a_2 \ldots a_nu$ where t is the part of the input that was already read and processed before we turned our attention to the computation, starting with ID I.

Also, u is the part of the input string that we will leave unprocessed after ID J, if indeed this ID is not terminal.

$$
\begin{aligned}
I = & tp_1a_1 \ldots a_nu \vdash ta_1p_2a_2 \ldots ta_nu \vdash ta_1a_2p_3a_3 \ldots a_nu \vdash \text{etc.} \\
& \vdash ta_1 \ldots p_ma_m \ldots a_nu \vdash ta_1 \ldots p_{m+1}a_{m+1} \ldots a_nu \vdash \text{etc.} \\
& \vdash ta_1 \ldots a_np_{n+1}u = J
\end{aligned}
$$

where above I used "$\ldots$" within an ID to denote not displayed *symbols* and used "etc." between IDs to denote not displayed *ID*s.

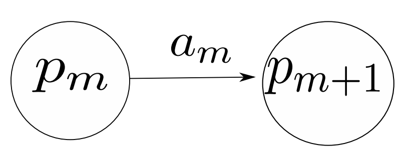
Note that each step (for any $m = 1, \ldots, n$)

$$ta_1 \ldots p_ma_m \ldots a_nu \vdash ta_1 \ldots p_{m+1}a_{m+1} \ldots a_nu$$

in the computation is *possible* (*valid*) IFF

$$\delta(p_m, a_m) = p_{m+1}$$

iff the graph has the edge



Having a *computation segment* —a *subcomputation*— due to an input *sub-stream* $a_1a_2\ldots a_n$ is **equivalent** to the *existence of a labeled path* — that we will aptly call a *computation path*— in the flow diagram M, from p_1 to state p_{n+1} —fig. below— *whose labels, concatenated from left to right, form the string $a_1a_2\ldots a_n$ that was processed (and "consumed") by the subcomputation*:

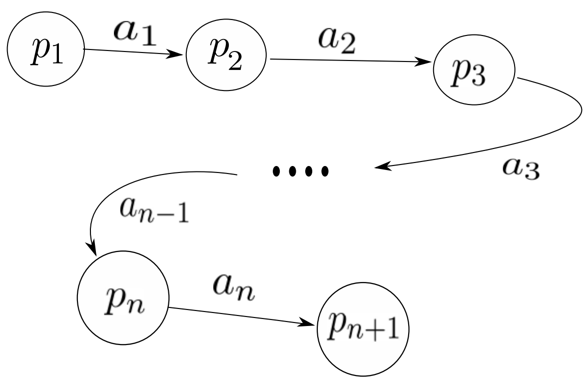


Figure 1: FA Computation Path

In particular, a string $x = a_1a_2\ldots a_n$ over the input alphabet belongs to $L(M)$ —the Language Accepted (Decided) by the FA M; cf. 0.1.1— *iff it is formed by concatenating the labels of a path such as the above, where $p_1 = q_0$ (start state) and p_{n+1} is* accepting*. In this case we have an* accepting path.

We see that the flowchart model of a FA is more than a static depiction of an automaton's "vital" parameters, Q, A, q_0, δ, F. *Rather,* all computations, *including* accepting computations, *are also* encoded *within the model as certain paths.*

□

Lecture #19, Nov.23

The last few paragraphs were important. Let as summarise:

0.2.11 Definition. (Graph acceptance) Let M be a FA of start-state "p_1" over the alphabet Σ.

Let $x = a_1 a_2 \ldots a_n$ be a string over Σ.

Then x is accepted by M —equivalently $x \in L(M)$ (cf. 0.1.1)— iff x is the label of a *computation path* in the graph version of M in the sense that x is obtained by concatenating the names $a_1, a_2, \ldots, a_n$ *OF THE EDGES* of said computation path (cf. Fig. 1) that starts at p_1 and ends at an accepting state p_{n+1}. The latter state has just scanned *eof* thus it caused M to halt. □

Armed with Definition 0.2.11, let us consider an example and shed more light on what exactly is *eof*.

0.2.12 Example.
Compilers, that is, **Systems Programs** that read programs written in a high level programming language like C and translate them into assembly language have several subtasks.

One of them is delegated to the so-called "scanner" or "token scanner" of the compiler and is the task of picking up variables from the program source. To "pick up" a variable, the scanner has to "*recognise*" that it saw one! Well, an automaton can do that!

Assume (as typically is the case) that the syntax of a variable is a string that

- *begins with a letter*

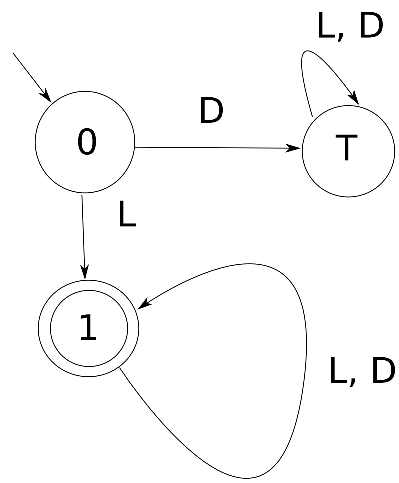
  and

- *continues with letters or digits*.

To simplify the example and not get lost in details, we denote the input alphabet of the automaton that we will build here $\Sigma = \{L, D\}$ where the symbol L stands for any **letter** (in real life, one uses the members of the set {A, B, C, . . . , Z; a, b,. . . , z}, sometimes augmented by some special symbols like $ and underscore).

Similarly the symbol D in our alphabet stands for **digit** (in real life, one has here the set of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$).

Using the characterisation of acceptance in 0.2.11, here is our design:



The only paths to state "1" (accepting) are labelled with L, followed by zero or more L and/or D in any order. That's the right syntax we want!

What is the role of state "T"?

T for trap! We do not want the first symbol of a variable to be other than L. So, if it is D we go to trap, never to exit from it (inputs L or D keep you in T, which is NOT an accepting state!)

► What if input is λ? We do not want that to be accepted either!

We are good since "0" —the start state— is NOT accepting. If λ was the string provided as input (not something starting with D), *then immediately 0 "sees" eof and halts. "0" being not accepting, λ is rejected!*

Finally, let us familiarise a bit more with *eof*.

This is not a unique end marker but is *context dependent*. In the context of variable names, in something like

$$LLLDDD + +$$

(in C++) the first + is *eof* as it is not in the alphabet of our scanner FA! Ditto if we had

$$LDDD := (LDLDDD + LLL)$$

in, say, Pascal. The first variable "$LDDD$" has ":" as *eof*. The second one "$LDLDDD$" has "+" as *eof*. The third one "LLL" has ")" as *eof*. □

0.2.13 Proposition. *If M is a FA, then $\lambda \in L(M)$ iff q_0 —the start state— is an accepting state.*

Proof. First, say $\lambda \in L(M)$.

By 0.2.11, we have a path labeled λ from q_0 to some accepting p.

Since there are no symbols in λ to consume *the only application of "read"* gave us *eof* and we are still at q_0. Thus $q_0 = p$ must be accepting.

Conversely, let q_0 is accepting.

The input stream looks like $\lambda¶$, where I generically indicated *eof* by "¶". This ¶ is scanned by q_0 and halts the machine right away.

But q_0 is accepting and λ is what was consumed before hitting *eof*. Thus λ is accepted: $\lambda \in L(M)$. □

0.2.14 Example.

Here is another example that we promised. Refer to Example 0.2.5. Consider the case where q_0 is accepting. Then the only possible acceptable strings x will have an even number of 1s —even parity— since to go from q_0 back to q_0 we need to consume a 1 going and a 1 coming.

But do we get an arbitrary string otherwise? Yes, since between any two consecutive 1s —and before the first 1 and after the last 1 we can consume any number of 0s.

Clearly, if q_1 was the accepting state instead, then we have an odd number of 1s in the accepting path since to end on q_1 as accepting state we need one 1, or three, or five, ... We add two 1s every time to leave q_1 and to go back. □

0.2.15 Remark. BTW, for any M, the set $L(M)$ —considered as a set of numbers since the symbols in the alphabet are essentially digits— is decidable!

The question $x \in L(M)$ is decided by the FA M itself: $x \in L(M)$ iff we have an accepting computation of M with input x. Cf. 0.2.11.

Wait! Is not decidability defined in terms of URMs? Yes, but an FA is a special case of a URM! □