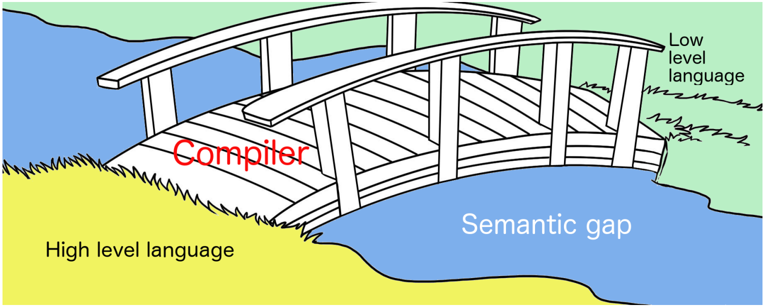
# Compilers and computer architecture: From strings to ASTs (1): finite state automata for lexing

Martin Berger [1]

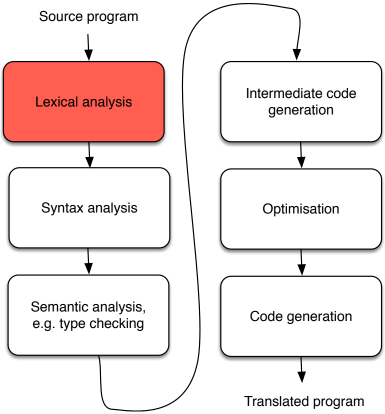
October 2019

[1] Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312

# Recall the function of compilers



# Plan for this week



Remember the shape of compilers?

We learned about regular expressions (REs). They enable us to specify simple language (finite and infinite).

The question we need to answer is: how to **decide**, given a string $s$ and a regular expression $R$, if $s \in lang(R)$?

We will later see that this is the main step towards an **algorithm** for lexing (tokenisation).

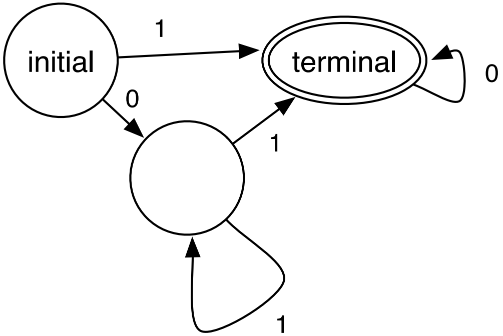
## Finite state automata

A finite state automaton (FSA) is an algorithm that, given a string over an alphabet *A*, answers with TRUE or FALSE. The strings that the FSA says TRUE to is the **language** of the FSA.

In other words, FSAs **decide** languages.

FSAs are easiest explained in pictures. Here is one with the alphabet $\{0, 1\}$

# Finite state automata



A word *w* is **accepted** by an FSA exactly if there is a path in the FSA from the initial state to a terminal state such that the edge labels we encounter on this path exactly spell the word *w*.

What language does the FSA above accept?

$$(1|01^+)0^*$$

## Finite state automata

A transition or edge $s \xrightarrow{a} t$ is to be understood as:

If the automaton is in state $s$ and reads ('eats') the character $a$ then it moves to state $t$.

If we are at the end of the input, and the automaton is in an terminal (also called accepting) state, the input string as a whole is accepted and in the language of the automaton.

If we cannot find a path that terminates at the end of the input, and the automaton is NOT in an accepting state, the input string as a whole is rejected and is NOT in the language of the automaton.

## FSA, formal definition

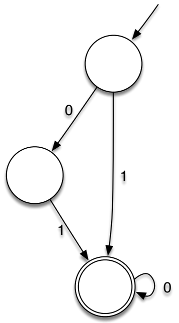
A **finite state automaton** (FSA) is a tuple $\mathcal{A} = (A, S, i, F, R)$ such that the following is true.

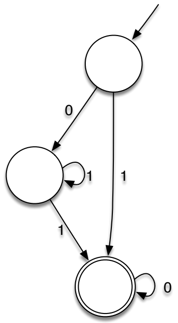
- $A$ is a finite set, called the **alphabet** of the automaton.
- $S$ is a non-empty finite set of **states**.
- $i \in S$ is the **initial state**.
- $F \subseteq S$ is the set of **terminal**, or **accepting states** of the automaton. **Note:** $F$ can be empty. (What happens then?)
- $R$ is the **transition relation**, i.e. it is a relation on states, characters and states. More formally, $R$ is a subset of $S \times A \times S$. We often write $s \xrightarrow{\alpha} t$ instead of $(s, \alpha, t) \in R$

We say $\mathcal{A}$ is **deterministic** if whenever $s \xrightarrow{\alpha} t$ and $s \xrightarrow{\alpha} t'$ then $t = t'$. Otherwise $\mathcal{A}$ is **non-deterministic**.

## FSAs, deterministic vs non-deterministic

Which one is deterministic, which on is non-deterministic?





The finite state automaton on the left is deterministic, that on the right non-deterministic. Each has one accepting state, indicated by double circles. Initial states are often drawn with an incoming arrow without source.

## FSAs, accepting a string

A string $(\alpha_1, ..., \alpha_n)$ is **accepted** by the automaton if and only if there is a path

$$i \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots s_{n-1} \xrightarrow{\alpha_n} s_n$$

where $i$ is the initial state, and $s_n$ is a terminal state. Note that the states $i, s_0, ..., s_n$ don't have to be distinct.

The **language** of an automaton $\mathcal{A}$ is the set of all accepted strings. We write lang($\mathcal{A}$) for this language.

## FSA examples

In class.

## FSAs vs REs

Why do we bother introducing FSAs when we've got REs to specify the lexical structure of a programming language?

Because we need an algorithm to **decide** membership in the language **specified** by the RE, and **convert** the input to a token list. FSA are (almost) algorithms. REs and FSAs are connected by the following amazing and surprising facts.

- For each regular expression $R$ over alphabet $A$, there is an deterministic FSA $F$ over $A$ such that $\text{lang}(R) = \text{lang}(F)$, and vice versa.
- For each non-deterministic FSA $F$ over alphabet $A$, there is an deterministic FSA $F'$ over $A$ such that $\text{lang}(F') = \text{lang}(F)$, and vice versa.

## Deterministic vs non-deterministic FSA: why bother?

An aside on the relationship between deterministic and non-deterministic FSAs: why bother at all with non-deterministic FSAs? Two reasons.

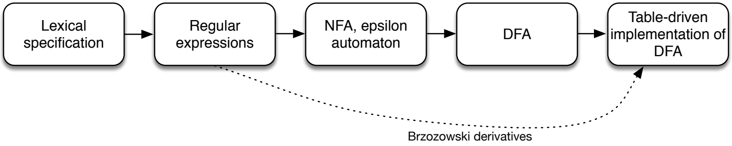
- Non-deterministic FSA are usually much **smaller** (fewer states) than the deterministic FSAs accepting the same language (often exponentially so: if the NFA has $n$ states, the DFA might have approximately $2^n$ states).
- Determinstic FSAs can be **implemented** on real machines. Question: Can non-deterministic FSAs be implemented (directly)?
- Non-deterministic FSAs can be **converted** to deterministic automata recognising the same language.

This is a familiar story: we look at something from two angles (1) convenient for humans vs (2) convenient for the machine.

## FSAs vs REs

Given that REs and FSAs can describe the same language, how can we get from an RE to an FSA?

Going straight from REs to deterministic FSAs is complicated. So we go there in several steps.



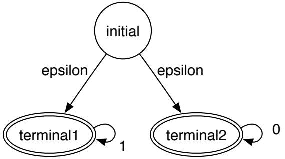
We are using $\epsilon$-automata which can be seen as a special case of NFAs. $\epsilon$-automata make the conversion from REs to Java implementations easier.

## $\epsilon$-automata

Formally, an **$\epsilon$-automaton with alphabet** $A$ is a (usually non-deterministic) FSA with alphabet $A \cup \{\epsilon\}$.

The definition of language $\epsilon$-automaton accepted by an $\epsilon$-automaton is slightly different from the definition for non-deterministic) FSAs.

What is $\epsilon$ for? We use $\epsilon$-labelled transitions $s \xrightarrow{\epsilon} t$ to move from state $s$ to state $t$, but **without consuming input.** This will be convenient later. What language does this $\epsilon$-automaton accept?



The language $0^*|1^*$ as a regular expression.

## $\epsilon$-automata

So, an **$\epsilon$-automaton with alphabet** $A$ is an FSA with alphabet $A \cup \{\epsilon\}$, **but** the language is different: the word $w$ over the alphabet $A$ is accepted by $\epsilon$-automaton $\mathcal{A}$ precisely when there is a word $w'$ over $A \cup \{\epsilon\}$ such that:

- If we remove all $\epsilon$ from $w'$ we obtain $w$.
- $w' \in \text{lang}(\mathcal{A})$ as a normal (i.e. walking any edge consumes the first character of the input string).

We write $\text{lang}_\epsilon(\mathcal{A})$ for the language of an $\epsilon$-automaton $\mathcal{A}$.

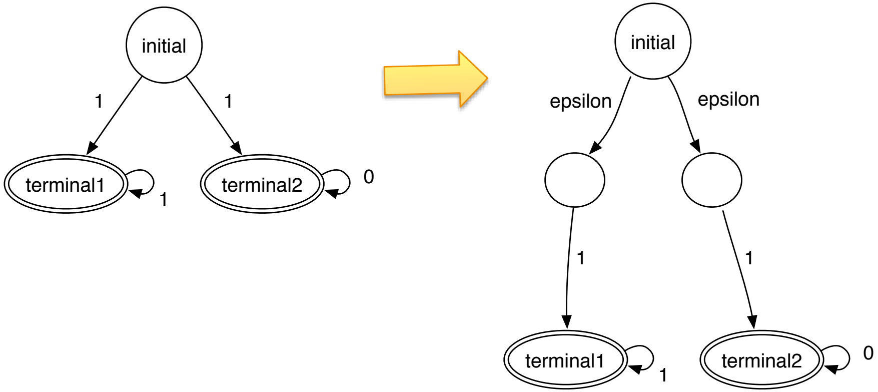
Example word: "**h e l** $\epsilon$ **l** $\epsilon$ **o**" gives us two chances to change state without consuming input and accept "**hello**".

So we have

$$\text{lang}_\epsilon(\mathcal{A}) = \{w \mid w' \in \text{lang}(\mathcal{A}), w \text{ is } w' \text{ with } \epsilon \text{ removed}\}$$

## $\epsilon$-automata are enough for non-deterministic FSA

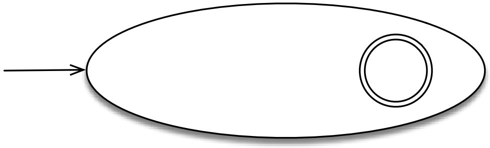
Non-determinism can always be translated to $\epsilon$-automata that are deterministic except for $\epsilon$-transitions.



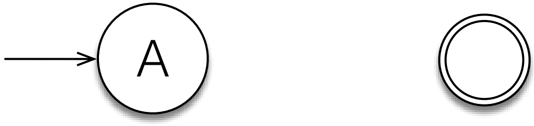
## Translation of REs to FSAs

We will translate every kind of RE ($\emptyset, \epsilon, R|R', ...$) into an FSA (an $\epsilon$-FSA to be precise).

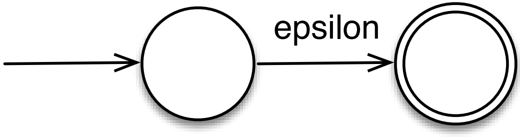
We don't need to details of each FSA in the translation, we will only be manipulating the initial and final state. All our translations have just one final state. We use the following notation to represent the FSAs arising in our translations.



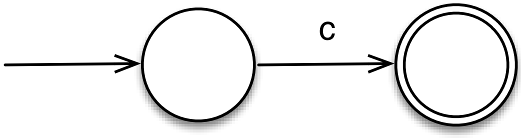
## Translation of $\emptyset$



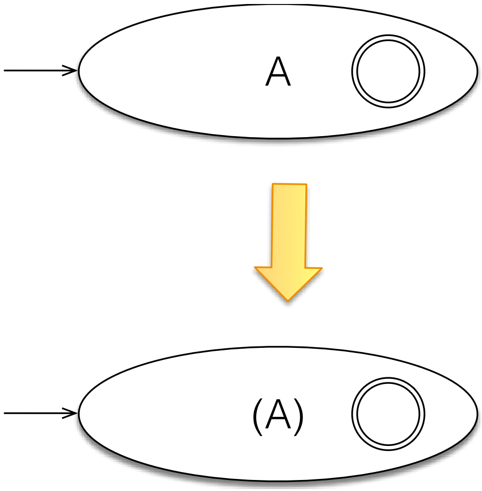
## Translation of $\epsilon$



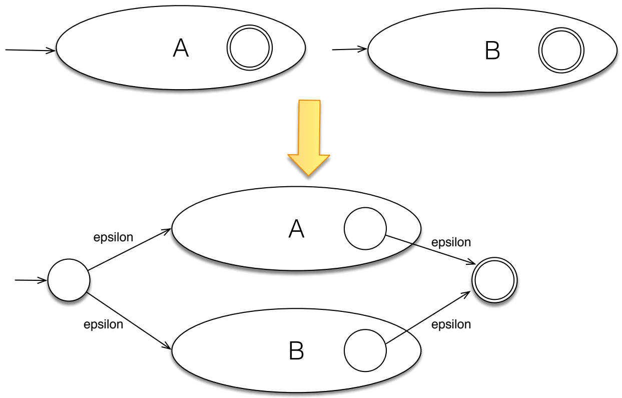
## Translation of $'c'$



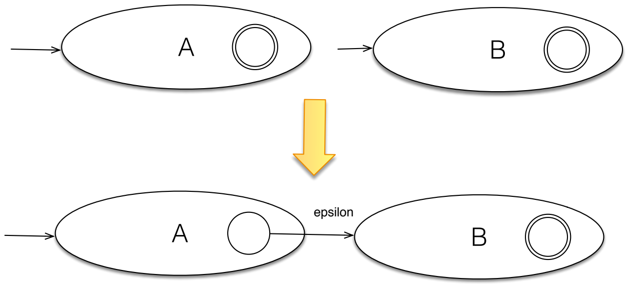
## Translation of (*A*)



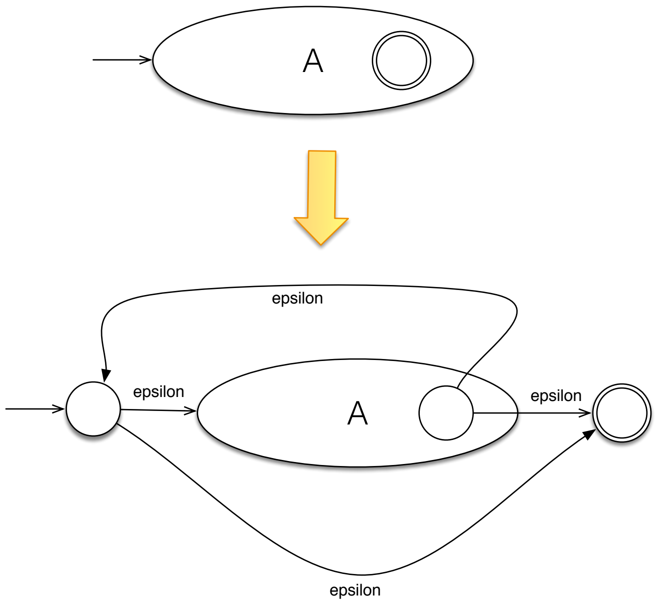
## Translation of $A|B$



## Translation of *AB*

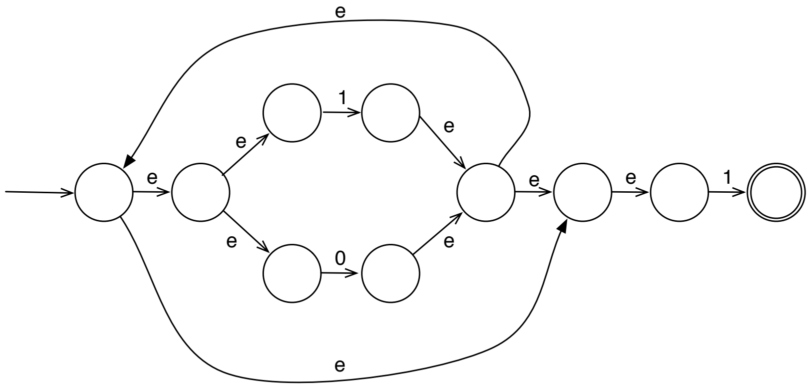


## Translation of $A^*$



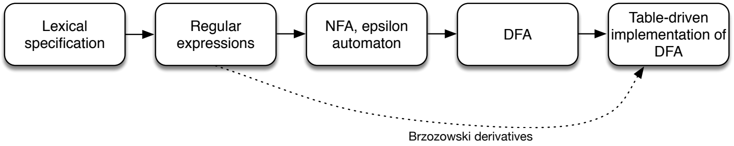
## Example translation

What’s the automaton that the RE $(1|0)^*1$ translates to?
(Writing *e* for $\epsilon$)



# From NFAs ($\epsilon$-automata) to DFAs

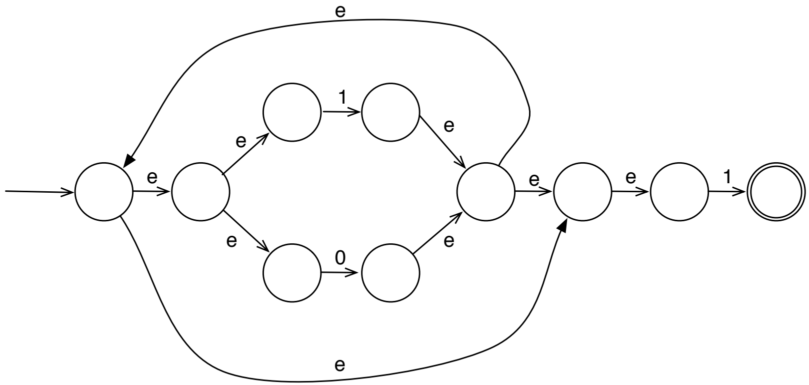
Remember the lexer construction pipeline?



Now we want to translate our NFAs ($\epsilon$-automata) to DFAs, because we can implement DFAs in e.g. Java (computers can't handle non-determinism).

## From NFAs ($\epsilon$-automata) to DFAs: $\epsilon$-closure

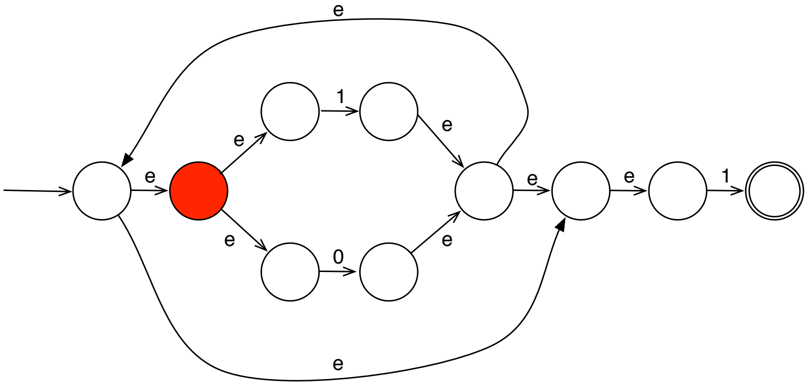
Consider the last example.

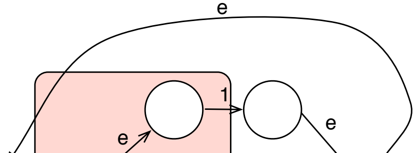


The $\epsilon$**-closure** of a set of states $S$ in an automaton is the set of all states reachable from a state in $S$ by 0 or more $\epsilon$-transitions.

## From NFAs ($\epsilon$-automata) to DFAs: $\epsilon$-closure.

Consider the last example.





# From NFAs ($\epsilon$-automata) to DFAs

Let $(A, S, i, F, \rightarrow)$ be an $\epsilon$-automaton ($A$ alphabet, $S$ states, $i \in S$ initial state, $F \subseteq S$ final states).

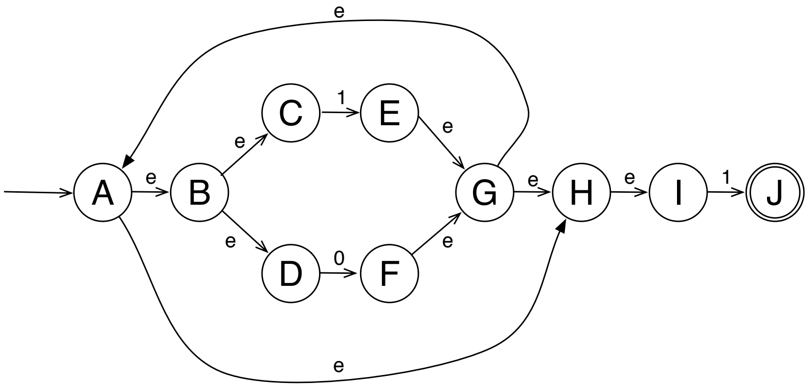
For each $a \in A$ and $X \subseteq S$ let $a(X) = \{y \in S \mid x \in X, x \xrightarrow{a} y\}$

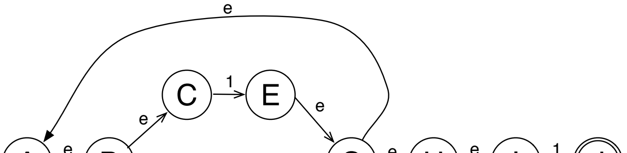
Now the corresponding DFA (accepting the same language) is given as follows.

- The new alphabet is $A$
- The new states are **all non-empty subsets** of $S$
- The new start state is the $\epsilon$-closure of $i$.
- The new final states are all non-empty sets $X \subseteq S$ such that $X \cap F \neq \emptyset$. (Why non-empty?)
- We have a new transition from $X$ to $Y$ with the label a exactly when $Y$ = $\epsilon$-closure of $a(X)$.

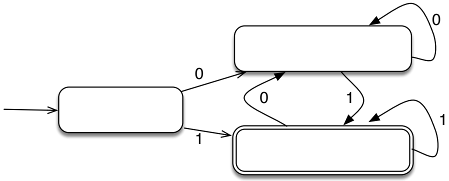
## From NFAs ($\epsilon$-automata) to DFAs

### The example





## From NFAs ($\epsilon$-automata) to DFAs



Check that the language of the new FSA is $(1|0)^*1$ as required.

## From NFAs ($\epsilon$-automata) to DFAs

Do you notice something? How many states does the new automaton have?

Consider our running example. It has 10 states. How many states would it DFA version have?

It has $2^{10} - 1 = 1023$ ...

This exponential blowup is an intrinsic problem of converting non-deterministic automata into deterministic ones. It has nothing to do with $\epsilon$-automata.

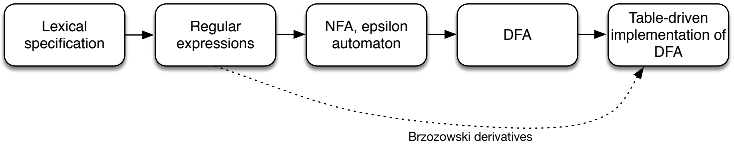
Fortunately in many cases, most of them are inactive and can be ignored. However in pathological cases, all states are needed.

## From NFAs ($\epsilon$-automata) to DFAs

This example shows that the translation in the naive form presented here is not particularly efficient: of the 1023 states it introduces, only 3 are needed (active). It is possible to improve the translation, so the inactive states disappear.

# Implementation of DFAs and NFAs

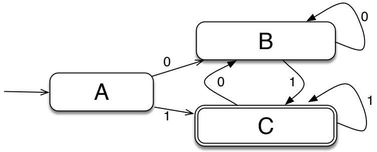
Remember the lexer construction pipeline?

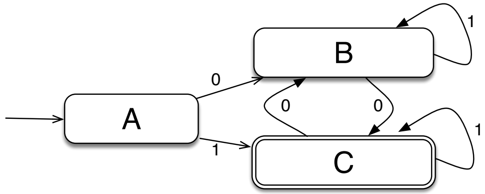


Now we want to translate DFAs into real programs e.g. Java.

## Implementation of DFAs

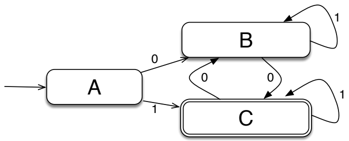
A DFA is naturally implemented as a 2-dimensional table (array) T. Columns are indexed by the alphabet, rows are indexed by the states. Array element at row *X* and character *c* stores the the next state in the automaton when starting in state *X* and consuming *c*.





|   | 0 | 1 |
|---|---|---|
| A | B | C |
| B | C | B |

## Implementation of DFAs



|   | 0 | 1 |
|---|---|---|
| A | B | C |
| B | C | B |
| C | B | C |

```
def scan ( input : Array [ Char ] )
                            : Boolean = {
   val table = ... // transitions
   var i = 0 // current character
   var s = A // current state
   val acceptingState = C
   while ( i < input.length ) {
      s = table [ s, input[i] ]
      i += 1 }
   return ( s == acceptingState ) }
```

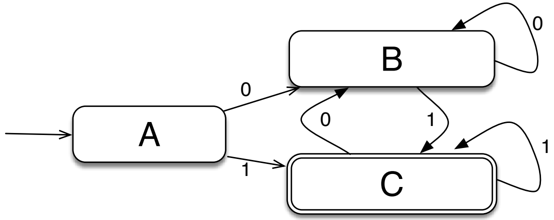
Question: what if one of the state lacks outgoing transitions on some labels? Answer: add artificial error states, and from the error state a transition back to itself for every character.

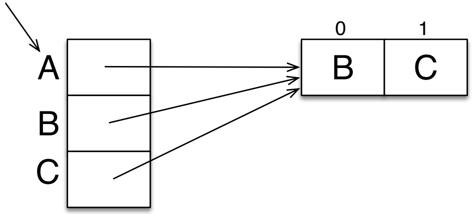
## Implementation of DFAs

This idea, using a 2-dimensional table to implement an FSA is fundamental. Most (all?) real-world implementations of REs, FSAs etc use variants of it. It is worth understanding well.

## Implementation of DFAs

Many rows in the array are identical (all in the example below, first and third row in the previous example). That is often the case in the implementation of lexers. We can save space by sharing rows (or columns):





## Outputting a token list

We have reached our intermediate goal: going from REs to algorithms that **decide** the language of the RE, i.e. respond with TRUE/FALSE for each input string. But in lexing we want a token list (or an error message). Fortunately, this is only a small variant of the decision problem.

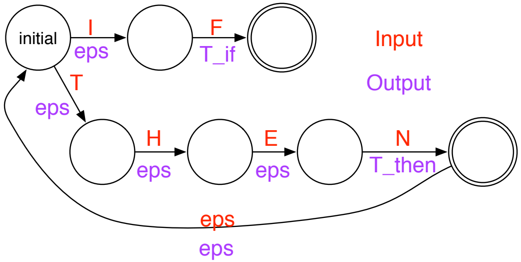
```
Hello ( 123 then ...
```

should yield a token list:

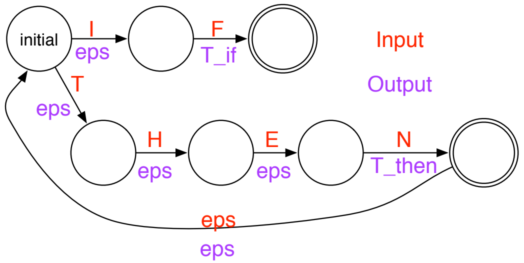
```
T_Ident ( "Hello" ),
T_Left Bracket,
T_Num ( 123 ),
T_Then,
...
```

# Mealy automata

We use **Mealy automata**, which is a variant of FSAs which have not only an input action, but also an output action. A picture says more than a 1000 words.



# Mealy automata



With a Mealy automaton, when we have a path

$$i \xrightarrow[u_1]{w_1} s_1 \xrightarrow[u_2]{w_2} s_2 \cdots s_{n-1} \xrightarrow[u_n]{w_n} s_n$$

whenever we accept (and consume) the input string $w_1 ... w_n$ we create an output $u_1, ..., u_n$. There are many variants, e.g. **Moore automata.**

## Mealy automata: implementation

We can implement Mealy automata by agumenting the 2-dimensional table with appropriate outputs that we accumulate as we consume the input string.

## Lexer generators

Lexers can be written by hand, but much easier to let the computer do that work. **Lexer generators** take as input an ordered list of REs (ordering gives priority, see below) together with **actions** (think Mealy automaton) associated with each RE, and returns a working lexer. Actions allow you to associate Java code with regular expressions. Examples: **Flex**, **JFlex**.

Lexer generator upside:

- Lexer generators produce very fast lexers
- Lexer generators isolate the compiler writer from having to worry about fast lexer implementations.

Lexer generator downside:

- Yet another thing to learn, and (like most software) tend to be badly documented.
- An expert can probably produce faster lexers than a generator.

## Implementing lexers using regular expression libraries

Modern programming languages often have elaborate regular expression libraries. They can be used for implementing lexers too. But you have to ensure things like “longest-match” and “keywords-first” heuristics.

Key disadvantage: regular expressions tend to be **slow**, so not suitable for industrial strength compilers. But OK for toy compilers.

## Summary

In first approximation, lexing works like this

- Write an RE for the lexemes of each token class, e.g. `Number = [0-9]`$^+$, `Keywords = ...`
- Construct a big RE, matching all lexemes for all tokens.

  ```
  R = Keywords | Identifier | Number | ...
  ```
- Construct an FSA (Mealy automaton) for *R*. Let a lexer generator do this work.

## Error handling in lexers

What if the lexer encounters a character in the input that does **not** match any RE defining the lexical level of the language? It's important for good compilers to return helpful error messages (not all compilers do this alas).

There's a neat way using regular expressions, longest match and priorities can also be used for error handling in lexers.

Use a RE that matches **any** character in the alphabet. Give this RE the lowest priority. Because it matches any character it will also always be a shortest possible match.

It catches anything that is not allowed by all previous REs. The output associated with this RE can be used for error messages.

## Conclusion

Lexer: takes a program as string, returns a list of tokens.

The point of lexing is to have a ROUGH classification of the input program that enables the next stage (parsing) to determine of the program is syntactically well-formed, and to construct the AST. Regular expressions and FSAs are convenient tools for implementation of lexers.

## The material in the textbooks

- Dragon Book: Chapter 2.6, Chapter 3.
- Appel, Palsberg: Chapter 2.
- “Engineering a compiler”: Chapter 2: especially sections 2.1 to 2.5.