# Compilers and computer architecture

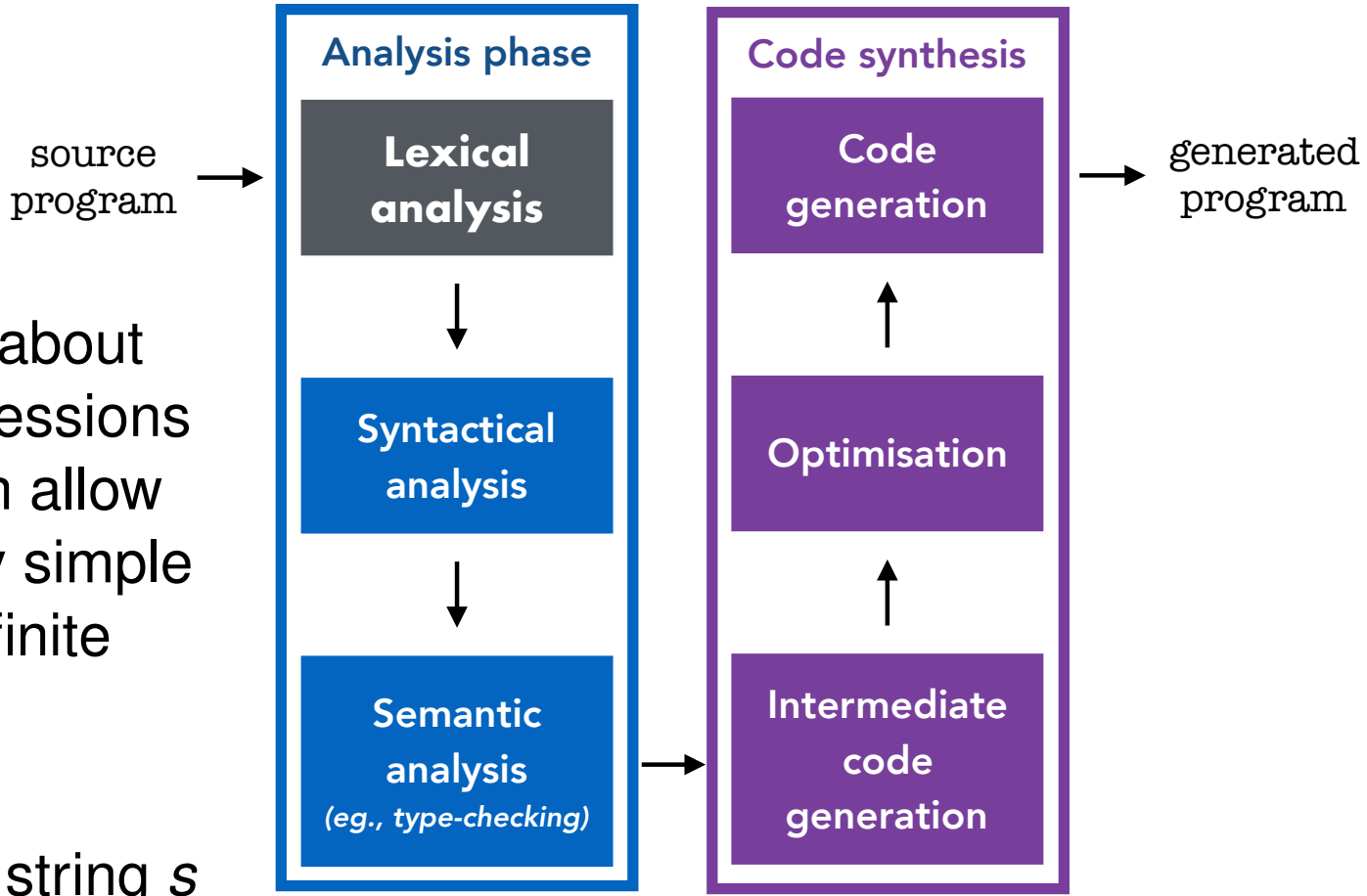
## 4. From strings to ASTs: lexing with finite state automata

**Ian Mackie**

Autumn 2022

# Plan for this week

Remember the shape of compilers:




We learned about regular expressions (REs), which allow us to specify simple languages (finite and infinite).

But, given a string $s$ and a regular expression $R$, how can we **decide** if $s \in \mathit{lang}(R)$?

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

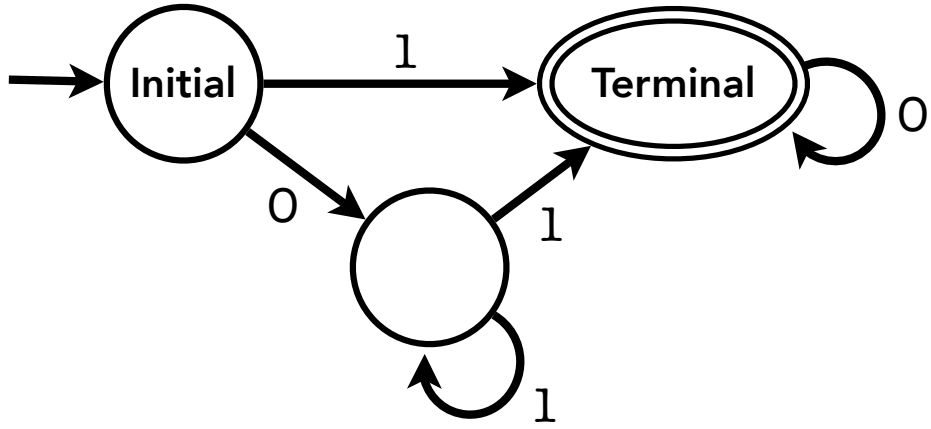
# Finite state automata

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




**Initial** states: drawn with an incoming arrow with no source.

**Terminal** (or **accepting**) states: drawn as double circles.

A word $w$ is **accepted** by an FSA,
exactly if there is a path* in the FSA such that:

- it starts at the initial state and ends at a terminal state
- the edge labels on this path spell the word $w$.

(* strictly speaking: we these are *walks* — we allow repeated nodes)

The language accepted by the FSA above: $(\mathtt{1} \mid \mathtt{01}^+)\mathtt{0}^*$

# Finite state automata

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

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

If we are at the end of the input, and the automaton is in a *terminal* (or *accepting*) state:

- the input string as a whole is accepted, and is in the language of the automaton.

If we *cannot* find a path from the initial state to a terminating state, which matches the input string:

- 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:

- $A$ is a finite set (the **alphabet** of the automaton)
- $S \neq \varnothing$ is a finite set (the set of **states** of the automaton)
- $i \in S$ (the **initial** state of the automaton).
- $F \subseteq S$ (the set of **terminal** / **accepting** states)
- $R \subseteq S \times A \times S$ (the **transition relation** of the automaton)

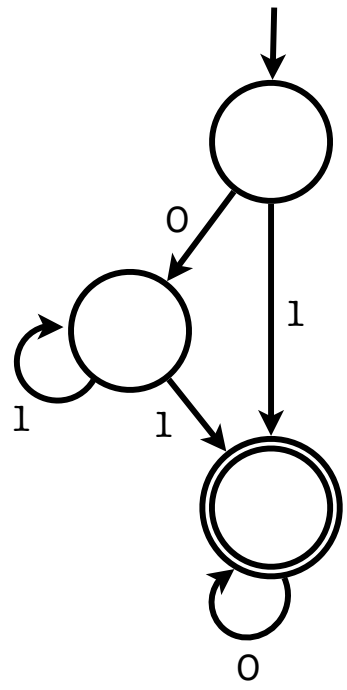
  — *i.e.*, $R$ is a relation on states, characters, and "successor" states.

  We often write "$s \xrightarrow{\alpha} t$", instead of $(s, \alpha, t) \in R$.

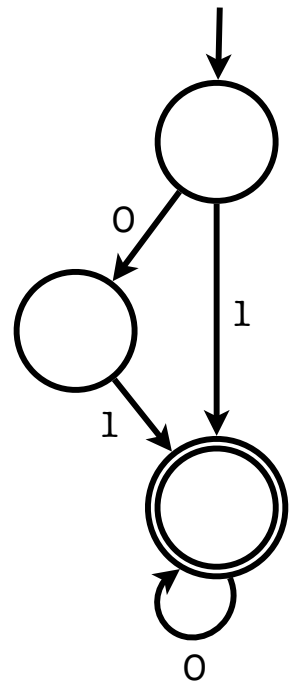
We say $\mathcal{A}$ is **deterministic** if for each $s$ and $\alpha$, there is *at most* one allowed transition $s \xrightarrow{\alpha} t$ (to any state $t$).
Otherwise $\mathcal{A}$ is **non-deterministic**.

# FSAs: deterministic vs non-deterministic

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




non-deterministic
(multiple transitions for **1**
on one of the nodes)




deterministic

Each has one accepting state.
(What are the languages accepted by these automata?)

# FSAs, accepting a string

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

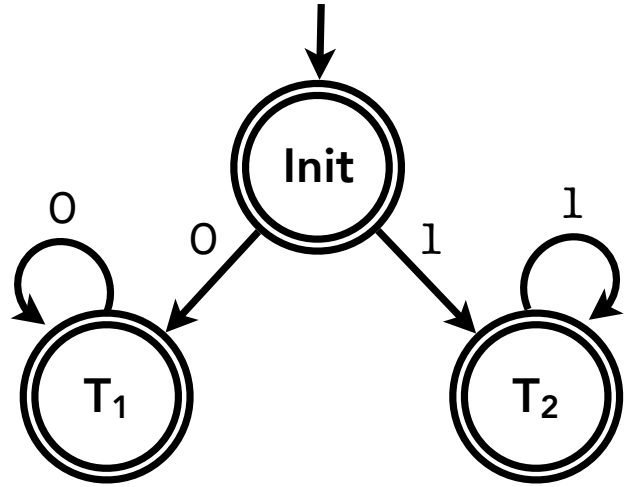
$$i \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{n-1}} s_{n-1} \xrightarrow{\alpha_n} s_n$$

where $i$ is the initial state, and $s_n \in F$ is an accepting state. (The states $i, s_1, ..., s_n$ 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.

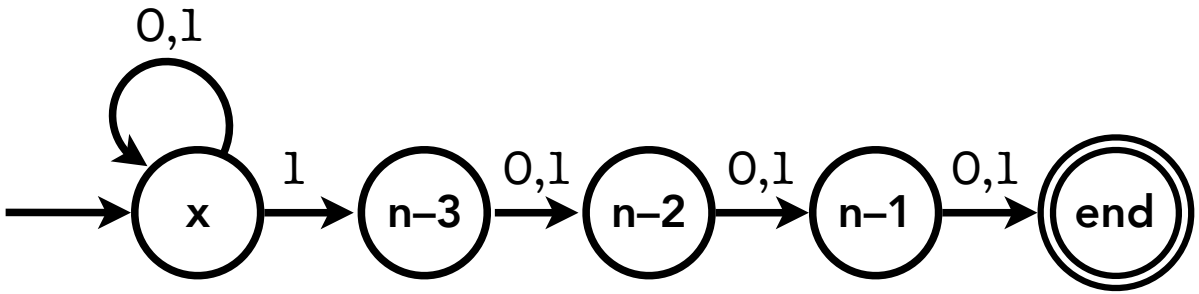
# FSA examples

- A deterministic finite state automaton (DFA) with three accept states.




What language does it accept? $\varepsilon \mid \mathtt{0}^+ \mid 1^+ = \mathtt{0}^* \mid \mathtt{1}^*$

- A nondeterministic finite state automaton (NFA).
  Arrows with $>1$ label are shorthand for multiple transitions.




What language does it accept? $(\mathtt{0}|\mathtt{1})^*\,\mathtt{1}\,(\mathtt{0}|\mathtt{1})(\mathtt{0}|\mathtt{1})(\mathtt{0}|\mathtt{1})$

# An enhancement: $\varepsilon$-automata

An enhancement on the definition of an FSE is the idea of an $\varepsilon$-automaton, which can help us to analyse regular expressions.

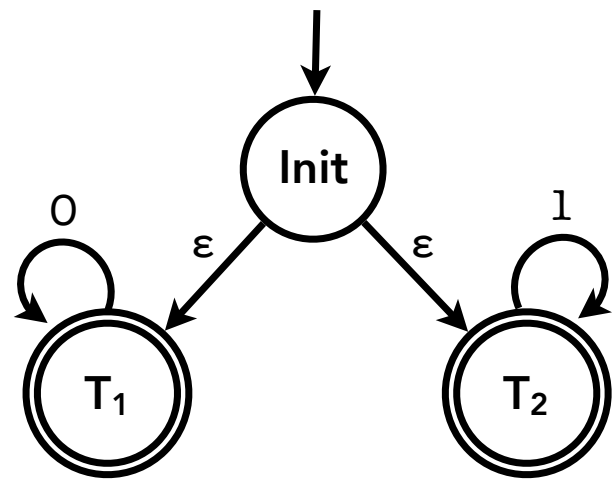
Formally, an **$\varepsilon$-automaton with alphabet** $A$ is an FSA (usually an NFA) with alphabet $A \cup \{\varepsilon\}$.

What is the point of introducing $\varepsilon$ ? — To define a different idea of the language accepted by this sort of FSA:

- We use $\varepsilon$-labelled transitions $s \xrightarrow{\varepsilon} t$ to move from state $s$ to state $t$, **without** consuming **any** input.

**Example.** What language does this $\varepsilon$-automaton accept?

**Answer:** the language of the regular expression $\texttt{0}^* \mid \texttt{1}^*$ .



# $\varepsilon$-automata

An **$\varepsilon$-automaton with alphabet** $A$ is an FSA with alphabet $A \cup \{\varepsilon\}$ . . . **but:** we also define a new notion of the language which it accepts.

- The word $w$ over the alphabet $A$ is accepted by $\varepsilon$-automaton $\mathcal{A}$, precisely when there is a word $x$ over $A \cup \{\varepsilon\}$ such that:
  - $x \in \text{lang}(\mathcal{A})$ according to the usual definition for FSAs (treating $\varepsilon$ as a normal character);
  - If we remove all $\varepsilon$ characters from $x$, we obtain $w$.

- We let $\text{lang}_\varepsilon(\mathcal{A})$ be the language of an $\varepsilon$-automaton $\mathcal{A}$, *i.e.*:

$$\text{lang}_\varepsilon(\mathcal{A}) = \left\{ w \;\middle|\; x \in \text{lang}(\mathcal{A});\ w \text{ is } x \text{ with every } \varepsilon \text{ removed} \right\}$$

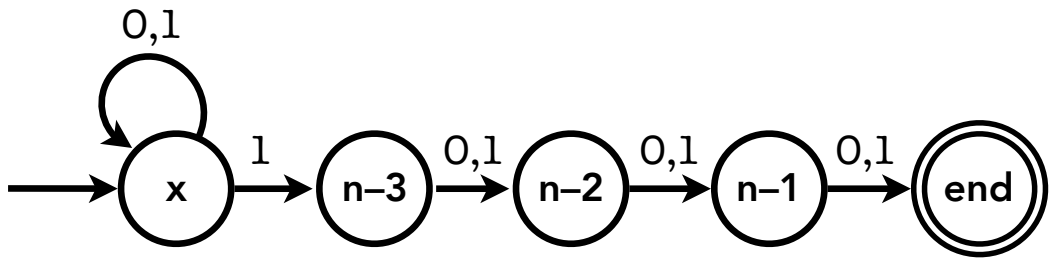
**But really:** this is just a stuffy way of saying that in an $\varepsilon$-automaton, you can make $\varepsilon$ transitions for free.

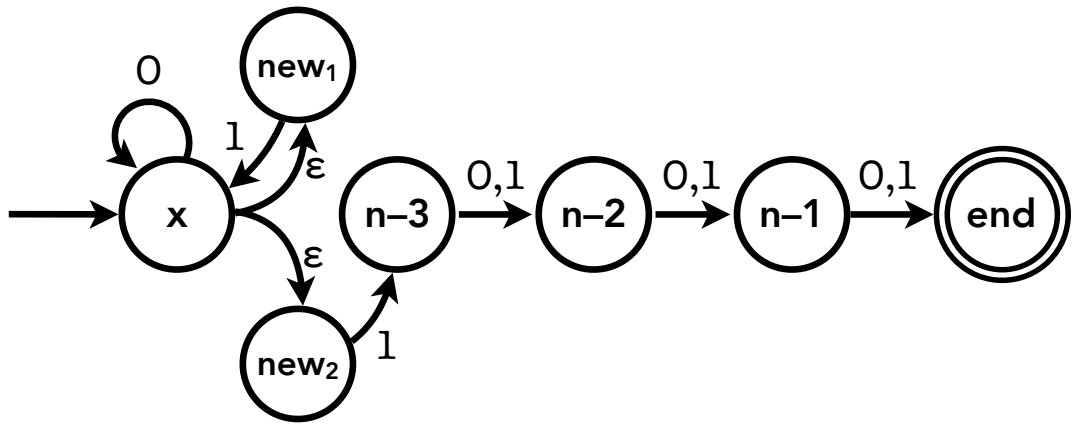
# $\varepsilon$-ifying NFAs

Non-determinism can be realised using only $\varepsilon$-transitions:

- NFAs can be turned into automata whose only non-deterministic transitions are $\varepsilon$-transitions.

**Example:**



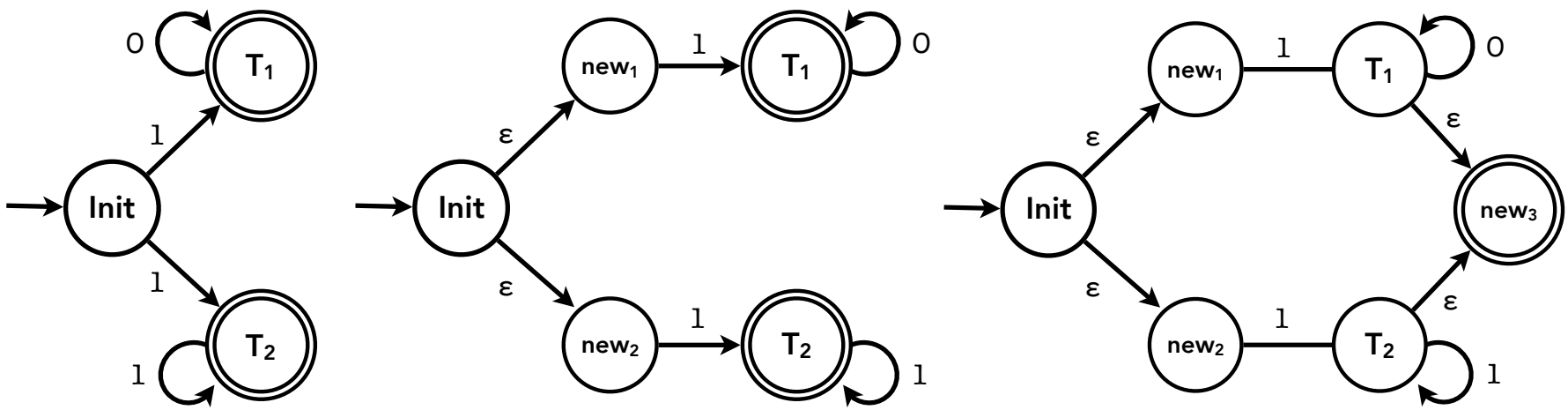



# $\varepsilon$-ifying NFAs

Non-determinism can be realised using only $\varepsilon$-transitions:

- NFAs can be turned into automata whose only non-deterministic transitions are $\varepsilon$-transitions.
- FSAs can be turned into automata which have at most one accepting state.

**Example:**



# FSAs vs REs

Why do we bother introducing FSAs, if we have REs to specify lexemes for programming languages?

Because we need an **algorithm**, not just a specification!

- To decide membership in the language specified by the RE
- To convert the input to a token list.

FSAs are *almost* efficient descriptions of algorithms; and DFAs in particular **are** efficient descriptions of algorithms.

REs and FSAs are closely related:

- For each regular expression $R$ over alphabet $A$, there is a DFA $\mathcal{A}$, such that $\text{lang}(R) = \text{lang}(\mathcal{A})$ — and vice versa.

# Deterministic vs non-deterministic FSA: why bother?

Every NFA can be turned into a DFA: that is —

- for each NFA $\mathcal{N}$ over alphabet $A$, there is a DFA $\mathcal{A}$ over the same alphabet, such that $\text{lang}(\mathcal{N}) = \text{lang}(\mathcal{A})$.

Furthermore: DFAs can be easily implemented in software, by software which tracks a program state in response to input.

(**Question:** can NFAs also be easily implemented?)

So, why bother at all with NFAs?

- NFAs are usually much **smaller** (fewer states) than a DFA accepting the same language

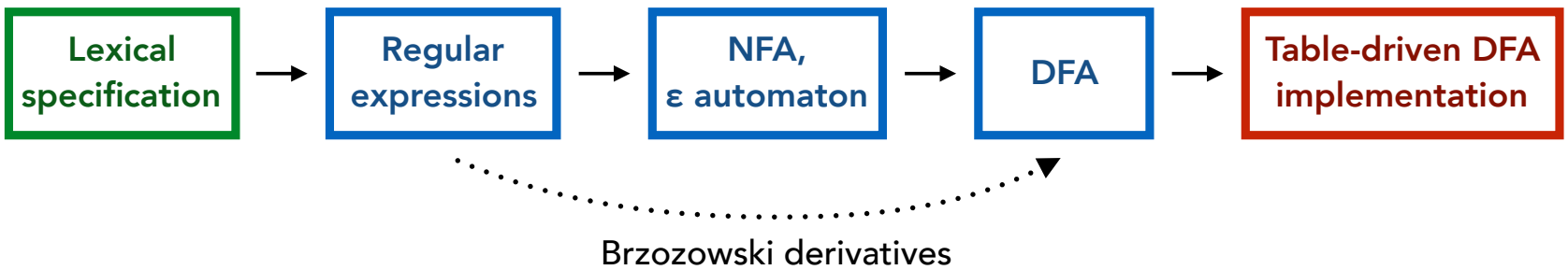
  — often exponentially so: if the NFA has $n$ states, the DFA might have nearly $2^n$ states!

- NFAs are a useful higher-level description to work with in some cases — as we will soon see.

# FSAs vs REs

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

We break down the construction in several steps:



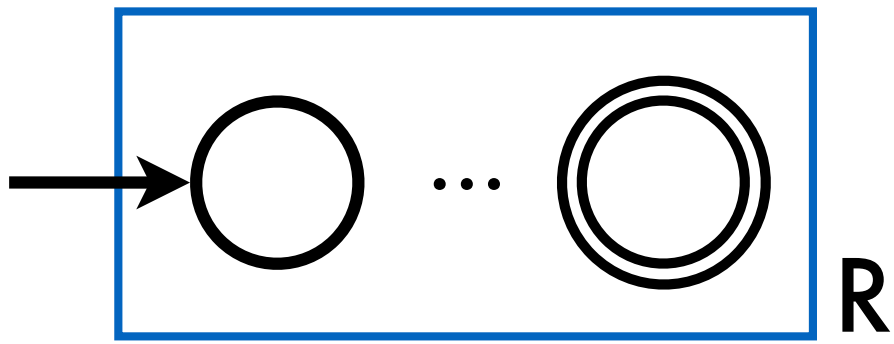

- Describe an NFA for a regular expression, by constructing NFAs recursively according to the description of the regular expression;
- Convert the NFA to a DFA.

**Modularisation!**

# Translation of REs to FSAs

To show how to obtain an FSA from a regular expression, we will translate every regular expression construction (*e.g.*, $\emptyset$, $\varepsilon$, $R \mid S$, $RS$, ...) into an $\varepsilon$-FSA with a final state.

We will describe this schematically, representing a typical $\varepsilon$-FSA for a regular expression $R$ as follows:

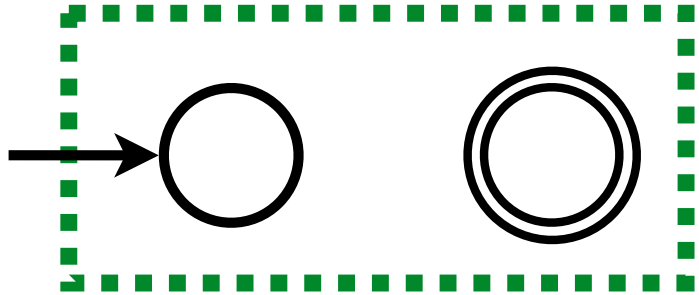



- A single accept state
- We don't care about the details of what's inside

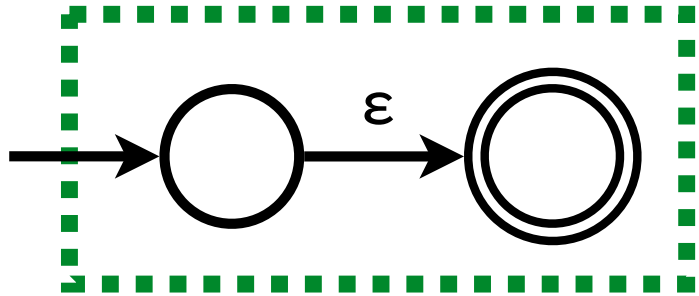
**Approach:** composing ("gluing") together $\varepsilon$-FSAs.

# Constructing $\varepsilon$-FSAs: base cases

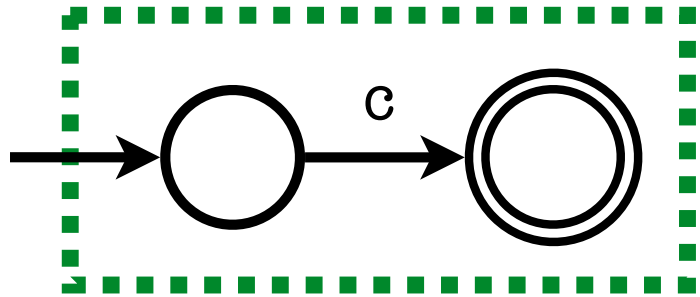
$\emptyset$



$\varepsilon$

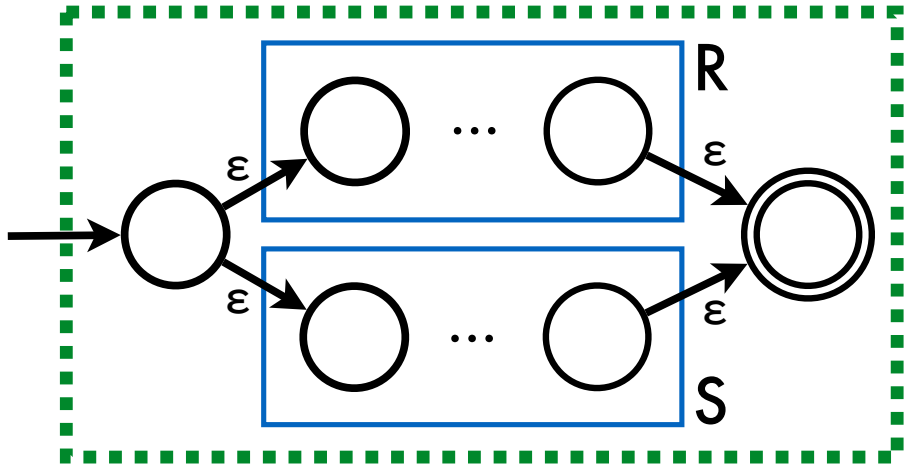



'c'

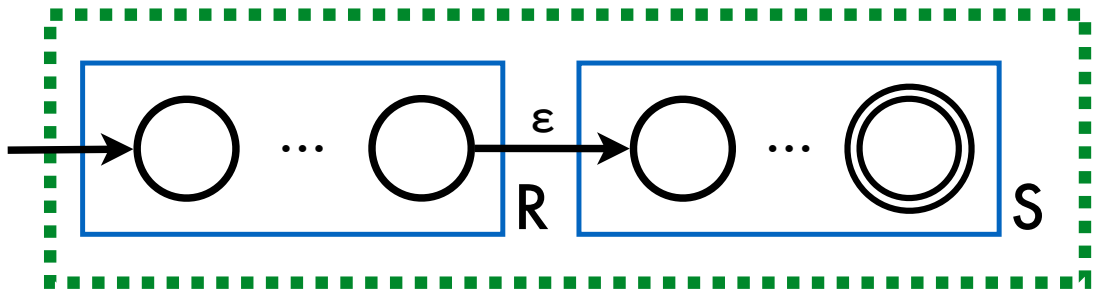


# Constructing $\varepsilon$-FSAs: recursive definitions

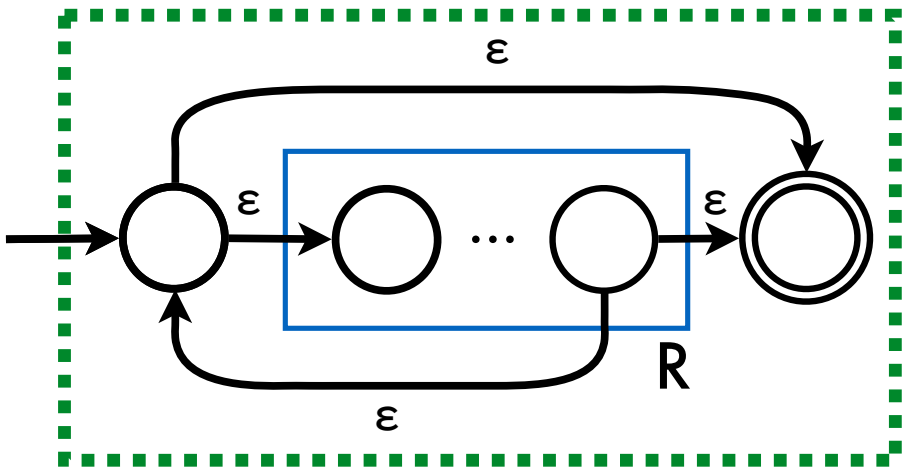
R | S




RS



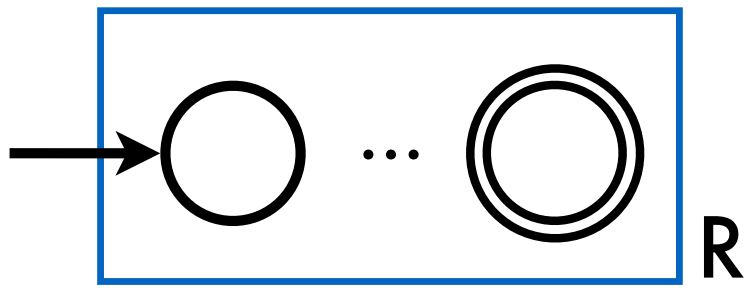

R*



# Constructing $\varepsilon$-FSAs: anything missing?

Parentheses!

(R)

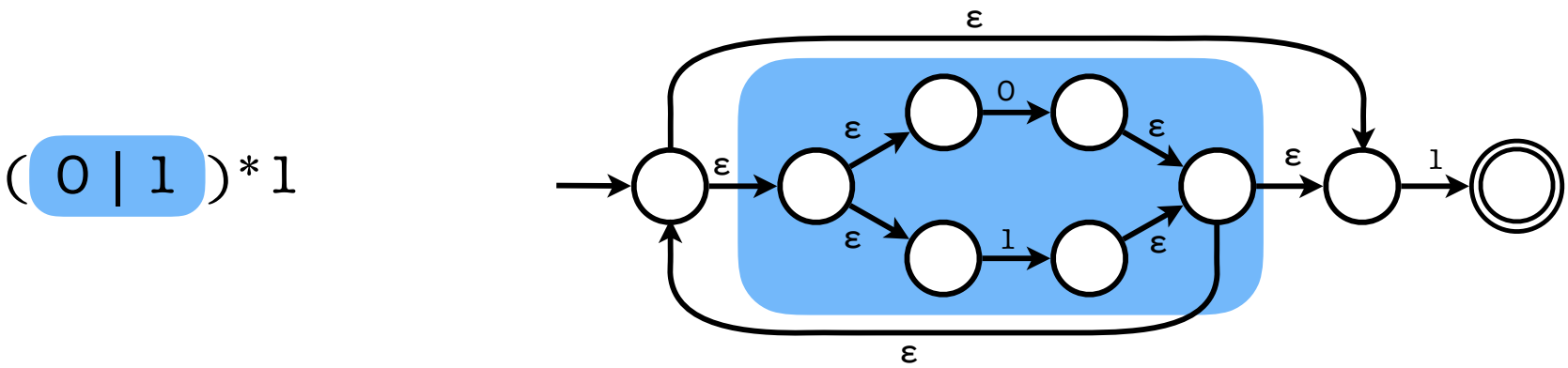



. . . nothing really changes!

- Parentheses aren't a significant recursive construction; they only control how REs are expressed / interpreted.

# Example translation

What's the automaton corresponding to the RE, `(0|1)*1`?

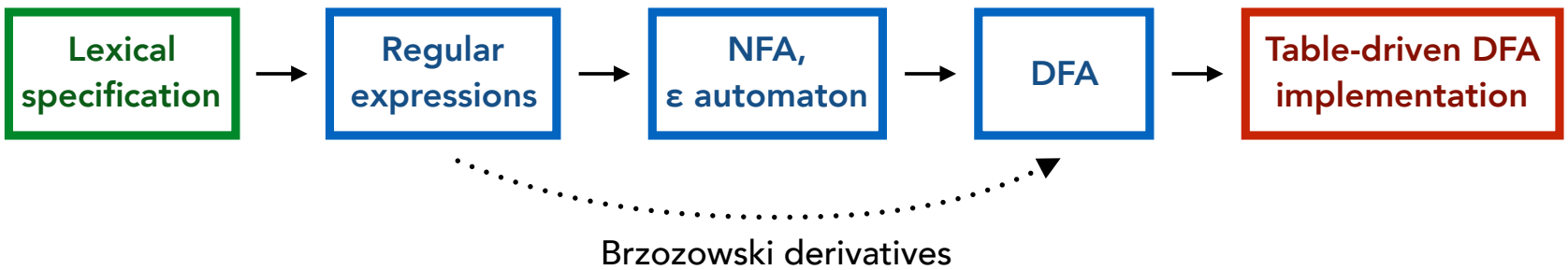



It's this one. Look closely to see how this automaton can be recursively constructed from the regular expression.

(For example: the hilighted part of the regular expression, corresponds to the hilighted part of the automaton.)

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

Remember the lexer construction pipeline:

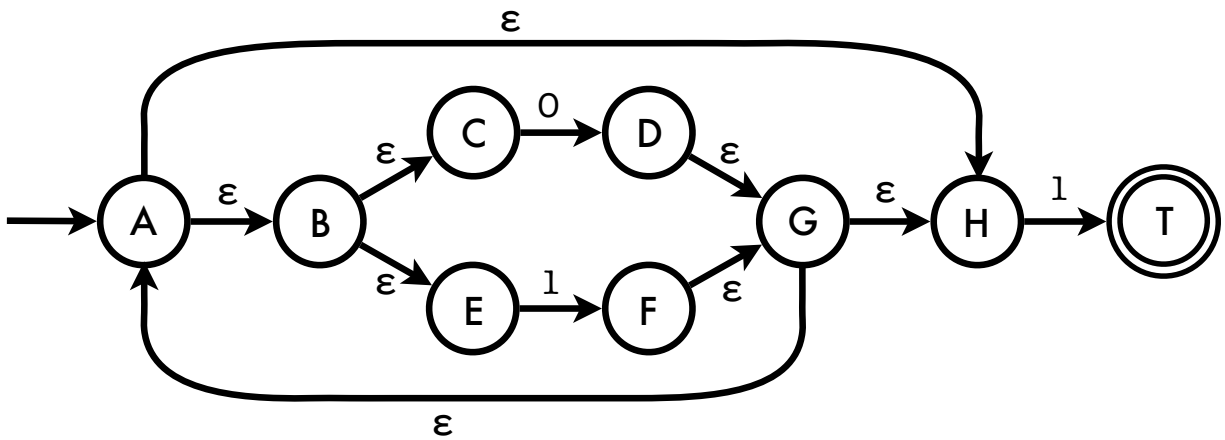



We next want to translate our NFAs ($\varepsilon$-automata) to DFAs

- We can implement DFAs straightforwardly, in *e.g.* Java
- Consider: what issues get in the way of representing NFAs more directly in code?

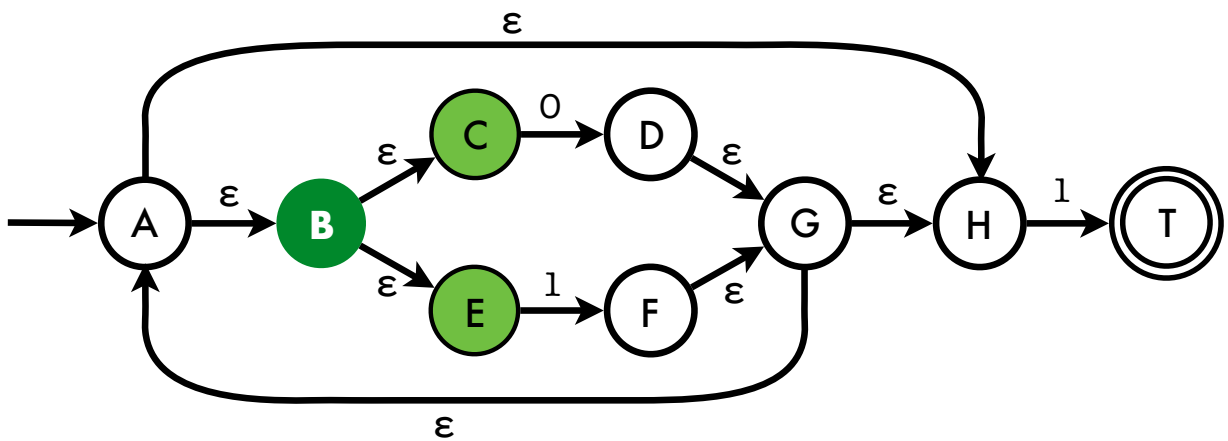
# From $\varepsilon$-automata to DFAs: $\varepsilon$-closure

Consider the earlier example of an $\varepsilon$-automaton:




The **$\varepsilon$-closure** $\mathrm{cl}_\varepsilon(X)$, of a set of states $X \subseteq S$ in an automaton, is the set of all states reachable from a state in $X$ by some number of $\varepsilon$-transitions (including no $\varepsilon$ transitions).

**Example:**



# From $\varepsilon$-automata to DFAs

Let $(A, S, i, F, R)$ be an $\varepsilon$-automaton $\mathcal{N}$:

- alphabet $A$, and states $S$
- $i \in S$ the initial state and $F \subseteq S$ the accept states
- $R$ is the transition relation, writing "$x \xrightarrow{\mathtt{a}} y$" for $(x, \mathtt{a}, y) \in R$. Extending this to **subsets** $X \subseteq S$, we write:

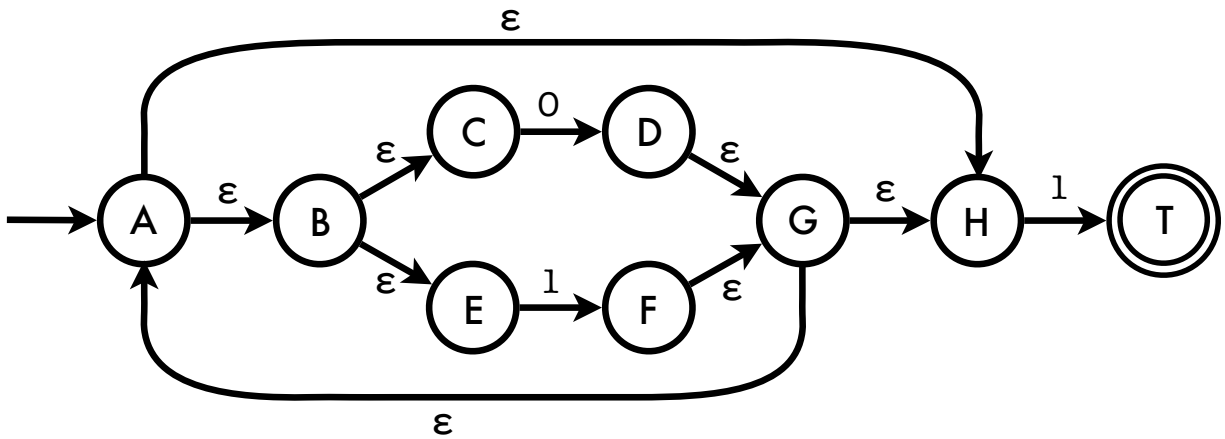
$$R(X, \mathtt{a}) = \left\{ y \in S \;\middle|\; x \in X;\ x \xrightarrow{\mathtt{a}} y \right\}$$

The corresponding DFA $\mathcal{A}$ which accepts the same language, is $(A, U, i_\varepsilon, T, R_\varepsilon)$, where:

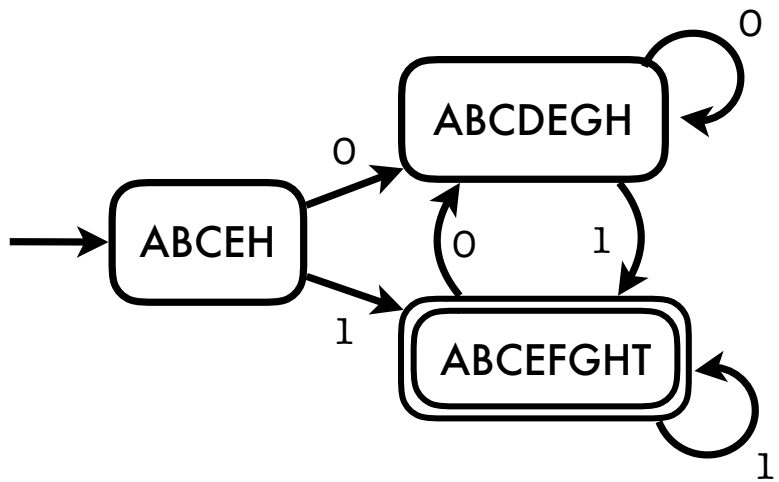
- alphabet is $A$; states $U$ are **all non-empty subsets** of $S$
- $i_\varepsilon = \varepsilon$-closure of the 'original' initial state $i$
- $T$ is the set of $X \in U$ which contain an element of $F$
- $(X, \mathtt{a}, Y) \in R_\varepsilon$ precisely when $Y = \mathrm{cl}_\varepsilon\big(R(X, \mathtt{a})\big)$
  - For $X \xrightarrow{\mathtt{a}} Y$ in $\mathcal{A}$, we have $y \in Y$ precisely when $y$ can be reached in $\mathcal{N}$ from some $x \in X$, by an $\mathtt{a}$-transition followed by a number of $\varepsilon$-transitions.

# From $\varepsilon$-automata to DFAs

From our running example,




we construct the following DFA (ignoring states not reachable from $i_\varepsilon = \{\text{A,B,C,E,H}\} = \text{“ABCEH”}$):

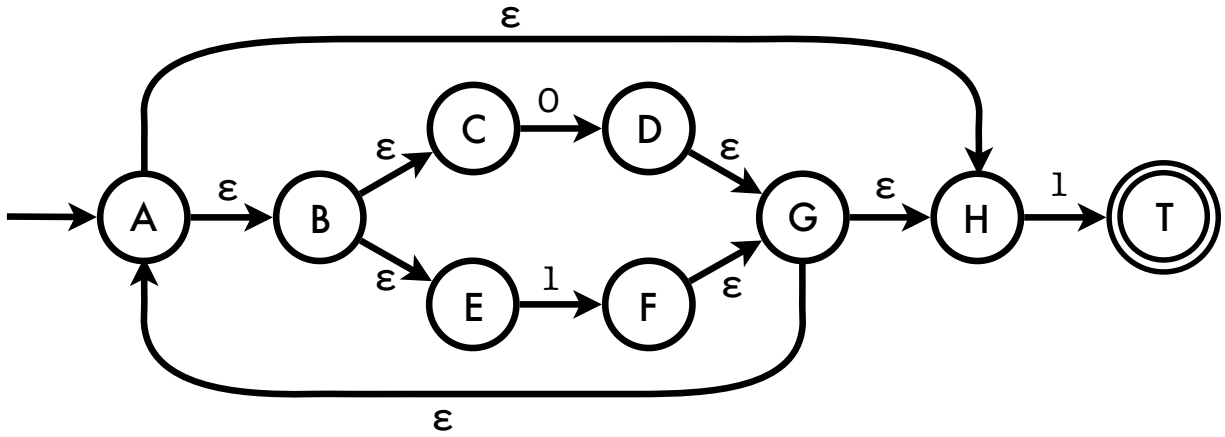



**Questions.**

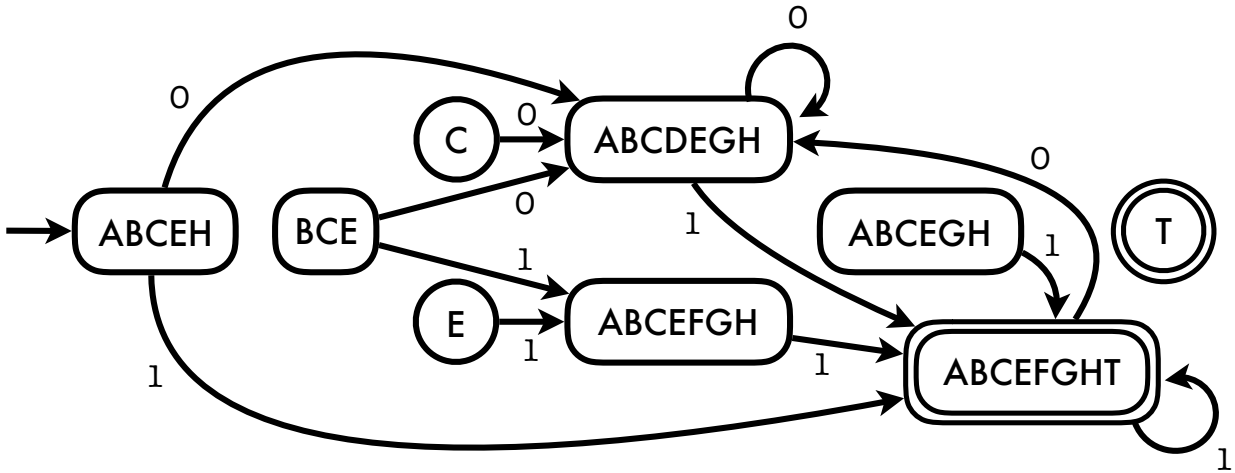
- Is the language of the new DFA $(\texttt{0} \mid \texttt{1})^* \, \texttt{1}$, as required?
- How many states does the new DFA **actually** have?

# From NFAs to DFAs: explosion of states

For our running example,




the corresponding DFA in fact has more than three states, *e.g.*:



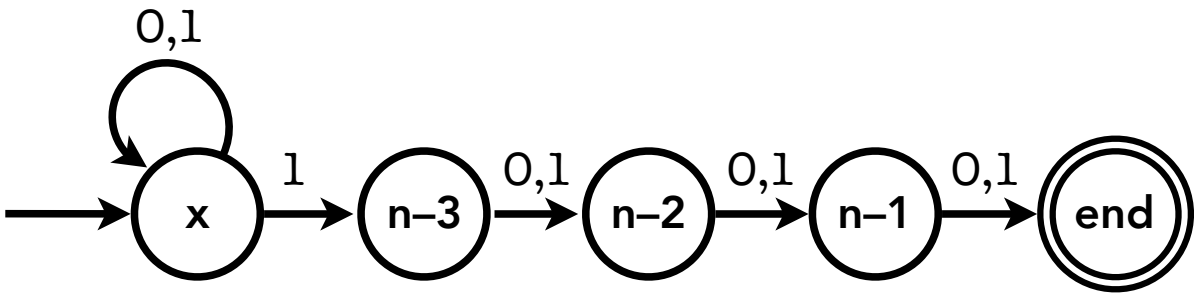

**Question.** How many states in total? … $2^9 - 1 = 511$ (!).

# From NFAs to DFAs: explosion of states

This example shows that the translation in the naive form presented here is not very efficient.

- In the running example above: only 3 of the 511 states are relevant (*i.e.*, can ever be reached).
- In many cases, we can ignore / eliminate a large number of inaccessible states

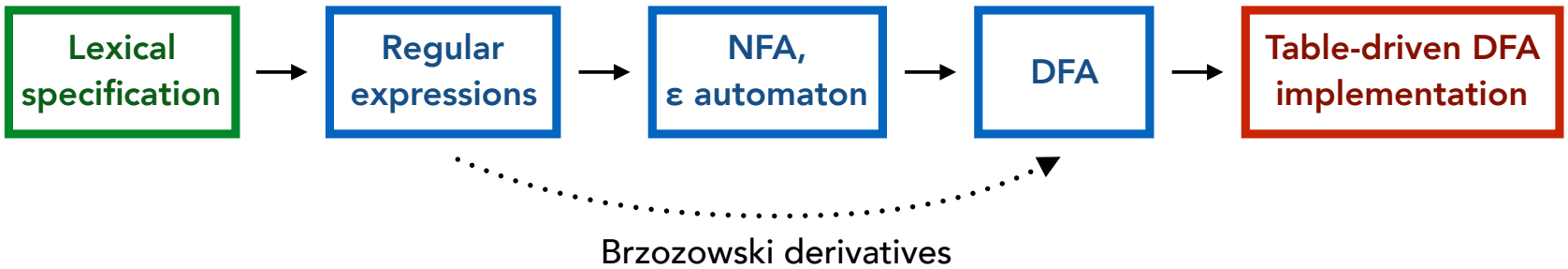
However: some $n$-state NFAs can only be simulated by $2^n - 1$ state DFAs, *e.g.*:




This exponential blowup is intrinsic to converting NFAs to DFAs.

# Implementation of DFAs by look-up tables

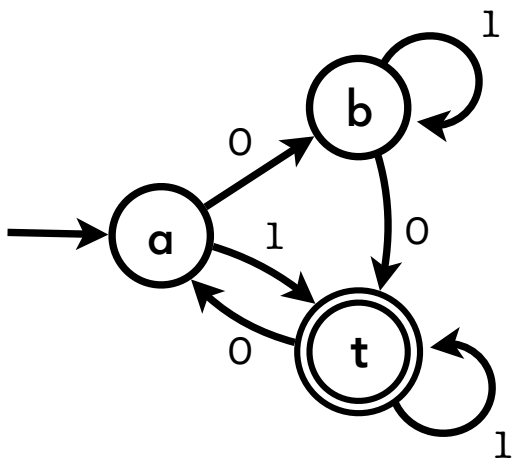
Remember the lexer construction pipeline:




We want to translate DFAs into real programs, *e.g.*, Java.

- Naturally implemented as look-up tables (*i.e.*, 2D arrays):
  - Columns indexed by the alphabet; rows indexed by states.
  - Array element at row $x$ and character `a` stores the the next state $y$ (*i.e.*, for which $x \xrightarrow{\texttt{a}} y$).

**Example:**




| | 0 | 1 |
|---|---|---|
| → a | b | t |
| b | t | b |
| t | a | t |

# Implementation of DFAs by look-up tables

| | 0 | 1 |
|---|---|---|
| → a | b | t |
| b | t | b |
| t | a | t |

```
public enum State {
  a, b, t;  // ...
}

class FSA {
  State s;
  State[][] table = {{b,t}, {t,b}, {a,t}};
  String input;

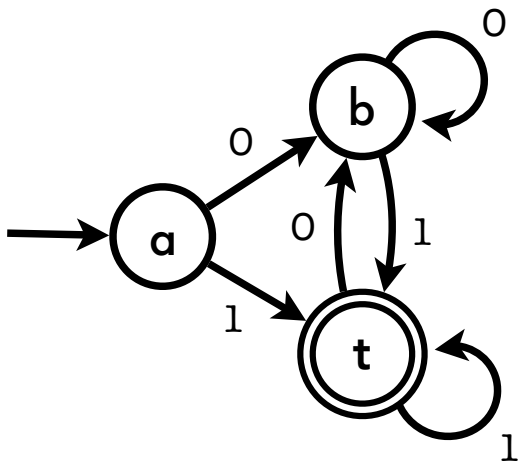
  public static State next (State s, char a) {
    char c = Character.getNumericValue(a);
    return table[s.index(), c];
  }

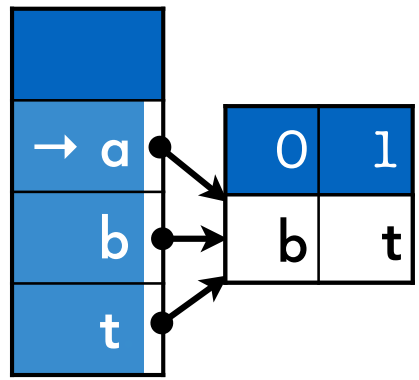
  public static void main (String[] args) {
    input = args[0];
    s = State.init();
    for (int j = 0; j < input.length(); j++)
      s = next(s,input[j]);
    return s.accept();
  }
}
```

# Implementation of DFAs by look-up tables

Many rows/columns may be identical (this happens often in practise). We can save space by sharing rows/columns.

**Example:**




**Question:** what if a state lacks a transition for some label?
**Answer:** add an ‘error’ state, and add a transition from this error state to itself for every character.

The idea of a look-up table to realise an FSA is important. Most (all?) real-world implementations of REs, FSAs, *etc.* use variants of it.

# 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` or `false` for each input string.

But: in lexing, we want a token list (or an error message)!

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

should yield a token list:

```
T_Ident("Hello"), T_lparen, T_Int(123),
T_Then, ...
```

This is only a small variant of the decision problem: we may use **Mealy automata**

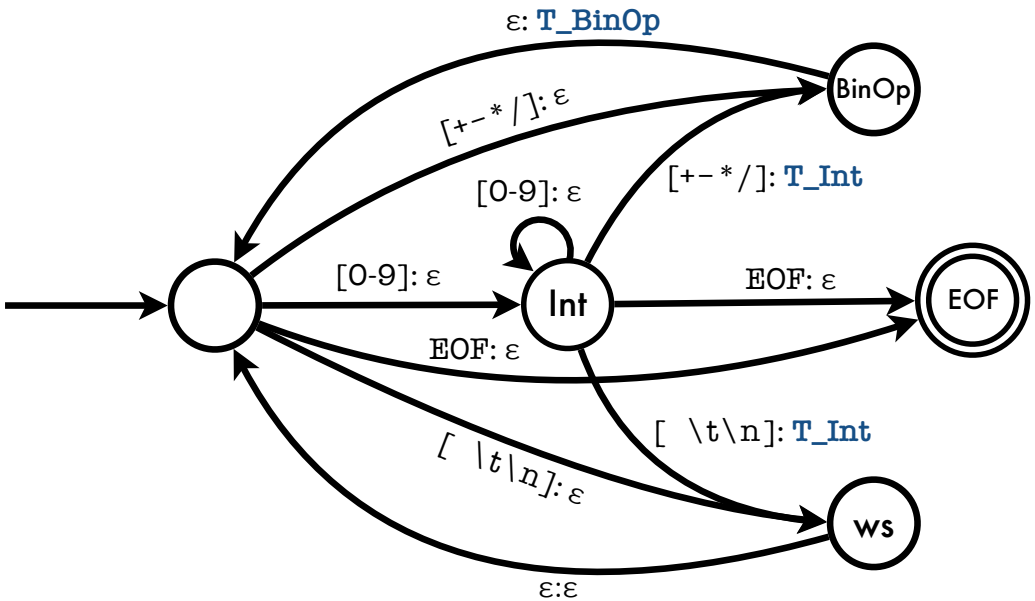
- a variant of FSAs which have an **output** action on each transition, as well as an input action.

# Mealy automata

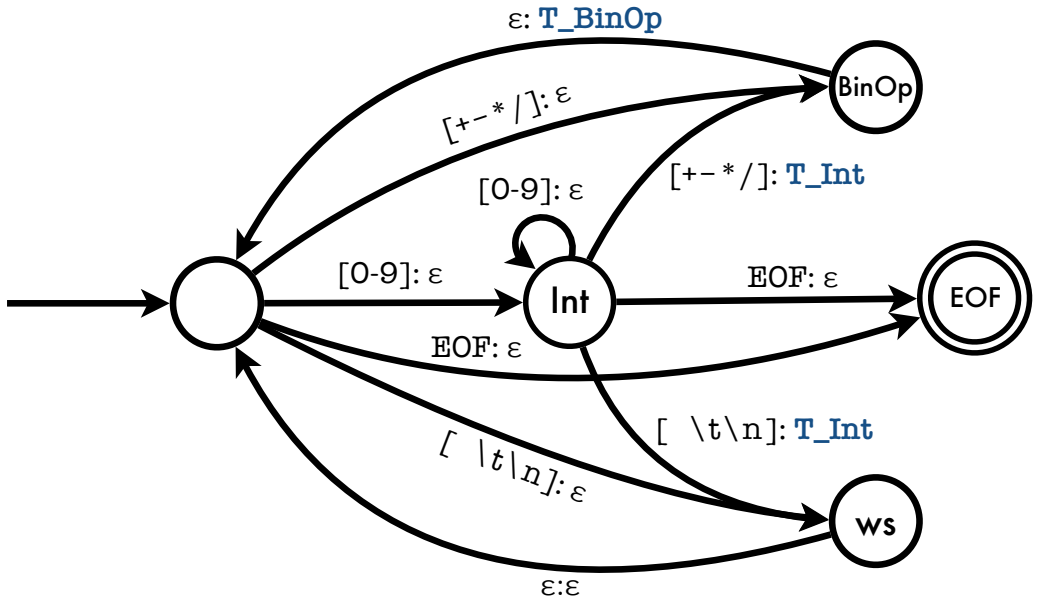
We use **Mealy automata**:

- a variant of FSAs which have an **output** action on each transition, as well as an input action.

**Example:** the following automaton reads a file over the alphabet $A = \{\texttt{0}, \ldots, \texttt{9}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \text{`` ''}, \texttt{\textbackslash n}, \texttt{\textbackslash t}, \texttt{EOF}\}$, and identifies integers and binary operators:



# Mealy automata




With a Mealy automaton, when we have a path

$$i \xrightarrow{\alpha_1 : u_1} s_1 \xrightarrow{\alpha_2 : u_2} s_2 \xrightarrow{\alpha_3 : u_3} \cdots \xrightarrow{\alpha_{n-1} : u_{n-1}} s_{n-1} \xrightarrow{\alpha_n : u_n} s_n$$

then we emit an output $u_1, \ldots, u_n$ whenever we accept (and consume) the input string $\alpha_1 \ldots \alpha_n$.

There are variations on this concept (*e.g.*, **Moore automata**).

# Mealy automata: implementation

We can implement Mealy automata by augmenting the look-up table with appropriate outputs that we accumulate as we consume the input string.

N.B. in order to eliminate $\varepsilon$ transitions, it may be necessary to emit more than one token per transition.

# Lexer generators

From a **practical** perspective: lexer generators (which rely on concepts such as Mealy automata) can help you to generate a lexer, from a specification of how it breaks down into tokens.

**Lexer generators** take as input:

- an ordered list of REs for the different token classes (ordering gives priority, see below)
- **actions** associated with each RE

— and returns a working lexer. The actions allow you to associate code with each token, to define how the lexer produces output.

(Lexers can be written by hand, but it is much easier to let the computer do that work.)

Examples: **Flex**, **BNFC**, **ANTLR**.

# Why not just use a regular expression library?

Modern programming languages often have elaborate regular expression libraries.

- These can be used to implement lexers, too!
- Be careful to ensure things like “longest-match” and “keywords-first” heuristics.

Key disadvantage: these tend to be **slow**

- Not suitable for industrial strength compilers.
- OK for toy compilers!

Another practical disadvantage:

- A little too easy to make mistakes!

# 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.

This will catch anything that is not allowed by all previous REs — the output associated with this RE can be used for error messages.

# Summary

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

- The point is to have a ***rough*** classification of the input stream, to simplify the task of constructing an AST.
- Regular expressions and FSAs are convenient tools for implementation of lexers.

Lexing works approximately as follows:

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

```
R = Keywords | Ident | Int | ...
```

- Construct an FSA (Mealy automaton) for $R$.
  Let a lexer generator do this work.

# 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.