G52MAL
Machines and Their Languages
Lecture 8

Equivalence of Regular Expression and Finite Automata

Henrik Nilsson

University of Nottingham

This Lecture (1)

This Lecture (1)

- We have seen three ways of formally describing potentially infinite languages:
  - Deterministic Finite Automata (DFA)
  - Nondeterministic Finite Automata (NFA)
  - Regular Expressions (RE)

This Lecture (1)

- We have seen three ways of formally describing potentially infinite languages:
  - Deterministic Finite Automata (DFA)
  - Nondeterministic Finite Automata (NFA)
  - Regular Expressions (RE)
- Because
  - a DFA is a special case of an NFA
  - any NFA can be converted into an equivalent DFA

  DFAs and NFAs describe the same ***class*** of languages: the ***Regular*** languages.

This Lecture (2)

So, what class of languages do the REs describe? Smaller? Larger? Completely different?

This Lecture (2)

So, what class of languages do the REs describe? Smaller? Larger? Completely different?

In fact:

- Regular Expressions describe the Regular Languages

This Lecture (2)

So, what class of languages do the REs describe? Smaller? Larger? Completely different?

In fact:

- Regular Expressions describe the Regular Languages
- Proof: interconversion between RE and FA

This Lecture (2)

So, what class of languages do the REs describe? Smaller? Larger? Completely different?

In fact:

- Regular Expressions describe the Regular Languages
- Proof: interconversion between RE and FA
- This lecture: conversion of RE to NFA

This Lecture (2)

So, what class of languages do the REs describe? Smaller? Larger? Completely different?

In fact:

- Regular Expressions describe the Regular Languages
- Proof: interconversion between RE and FA
- This lecture: conversion of RE to NFA

Will start by a motivating example; time permitting will look at another application: scanners.

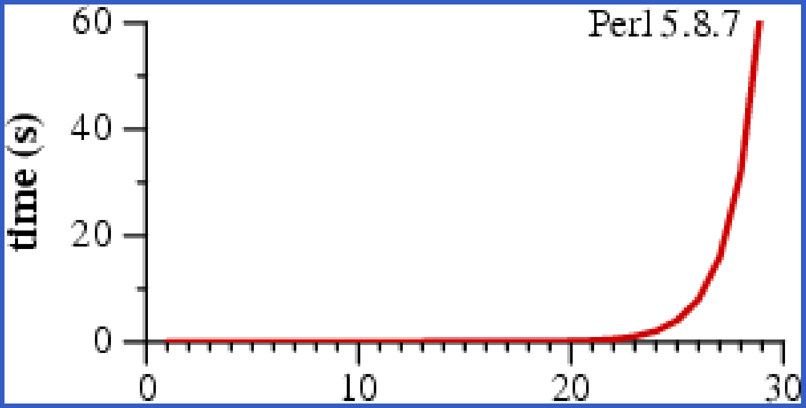
Applications (1)

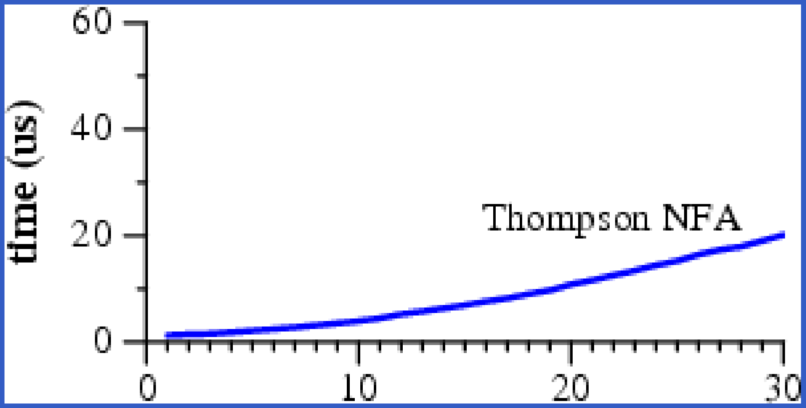
RE to NFA conversion has important practical applications.
The following is a very nice, practically oriented article you should be able to fully appreciate based on what you have learned in G52MAL thus far:

> Russ Cox. *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)*, January 2007.
>
> `http://swtch.com/~rsc/regexp/regexp1.html`

Applications (2)

Underlying message: if you're ignorant about CS theory, your code can perform really poorly. Example from the paper:





Time to match $(\mathbf{a} + \epsilon)^n \mathbf{a}^n$ against a^n

Note difference of time scale: 60 s vs. 60 μs!

Applications (3)

To quantify:

- Thompson NFA implementation a ***million*** times faster than Perl when running on a 29-character string.
- Thompson NFA handles a 100-character string in under 200 microseconds; Perl would require over 10^{15} years.

Recap: Syntax of Regular Expressions

1. $\emptyset$ is an RE
2. ϵ is an RE
3. For all $x \in \Sigma$, $\mathbf{x}$ is an RE
   (Handwriting convention: $\underline{x}$ is an RE)
4. If E and F are REs, so is $E + F$
5. If E and F are REs, so is EF
6. If E is an REs, so is E^*
7. If E is an REs, so is (E)

These are ***all*** regular expressions.

Recap: Semantics of Regular Expr.

1. $L(\emptyset) = \emptyset$
2. $L(\epsilon) = \{\epsilon\}$
3. For all $x \in \Sigma$, $L(\mathbf{x}) = \{x\}$
4. $L(E + F) = L(E) \cup L(F)$
5. $L(EF) = L(E)L(F)$
6. $L(E^*) = L(E)^*$
7. $L(\ (E)\ ) = L(E)$

Converting RE to NFA (1)

We are going to detail a “Graphical Construction” for converting an RE to an NFA that is suitable for carrying out by hand.

It can be further refined into a fully formal algorithm: see the lecture notes for details.

Converting RE to NFA (2)

Specification:

Let $N(E)$ denote the NFA that results by applying the graphical construction to an RE E. Then the following equation must hold:

$$L(E) \;=\; L(N(E))$$

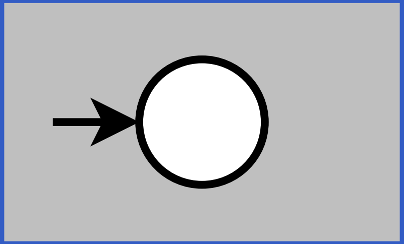
(Note that L is ***overloaded***: the language of an RE to the left, the language of an NFA to the right.)

We proceed case by case according to the structure of the syntax of REs.

RE to NFA, Case $\emptyset$

Recall: $L(\emptyset) = \emptyset$

$N(\emptyset)$:



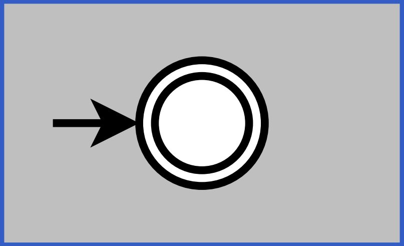
Note: $L(N(\emptyset)) = \emptyset = L(\emptyset)$; specification satisfied in this case.

Note: States are given without names for simplicity. Suffice as construction is graphical; states to be named at the end.

RE to NFA, Case ϵ

Recall: $L(\epsilon) = \{\epsilon\}$

$N(\epsilon)$:

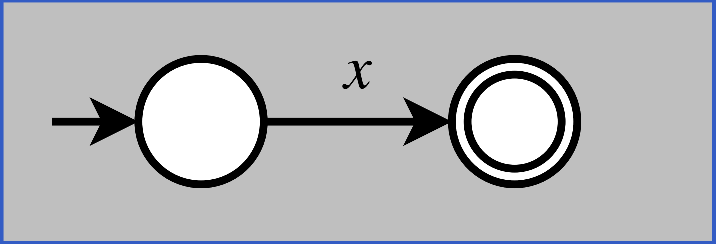


Note: $L(N(\epsilon)) = \{\epsilon\} = L(\epsilon)$; specification satisfied in this case.

RE to NFA, Case x for $x \in \Sigma$

Recall: For each $x \in \Sigma$, $L(\mathbf{x}) = \{x\}$

$N(\mathbf{x})$:

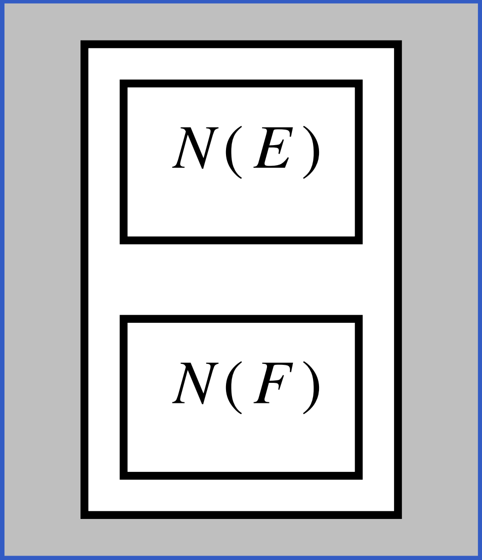


Note: $L(N(\mathbf{x})) = \{x\} = L(\mathbf{x})$; specification satisfied in this case.

RE to NFA, Case $E + F$ (1)

Recall: $L(E+F) = L(E) \cup L(F)$

$N(E+F)$:



The NFAs $N(E)$ and $N(F)$ in parallel. The initial states of $N(E+F)$ are the union of the initial states of $N(E)$ and $N(F)$.

RE to NFA, Case $E + F$ (2)

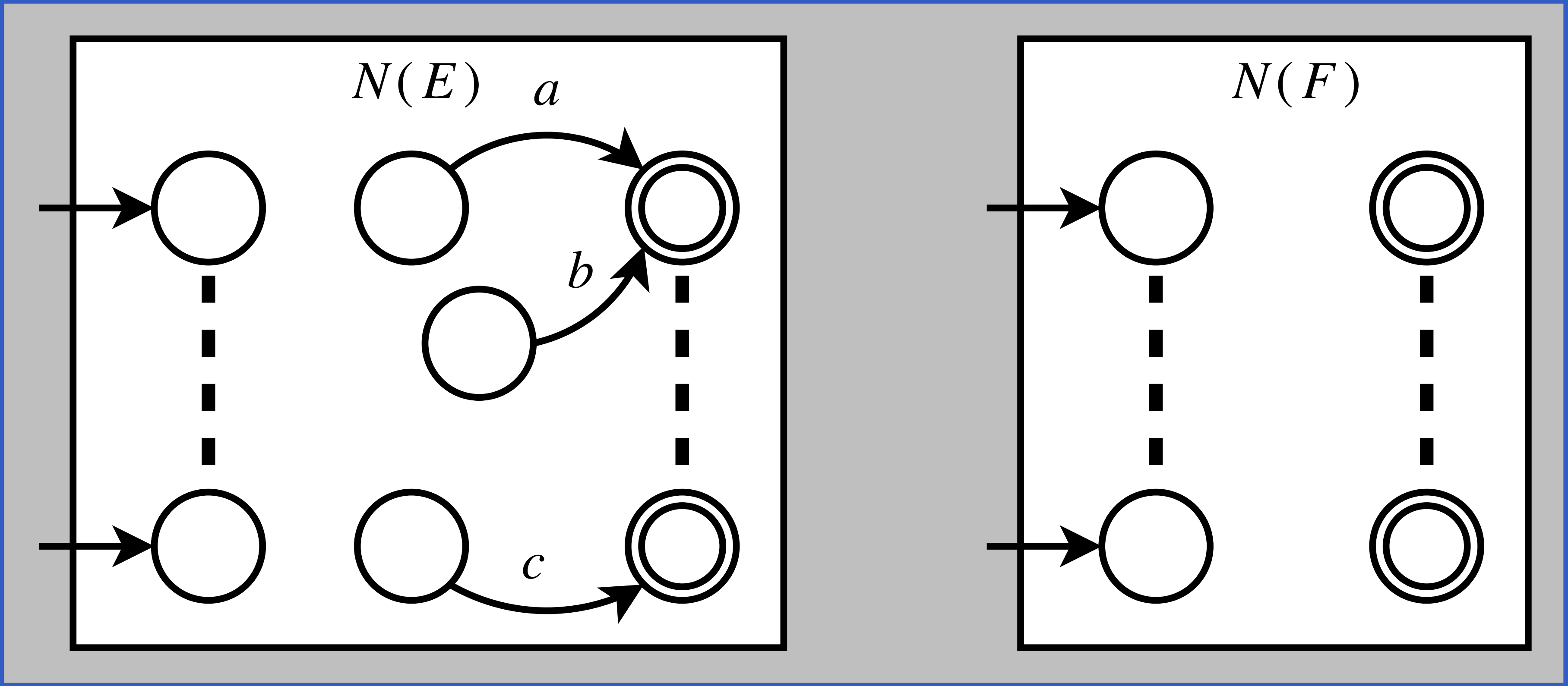
Note: Assuming specification holds for E and F,

$$
\begin{aligned}
L(N(E+F)) &= L(N(E)) \cup L(N(F)) \\
&= L(E) \cup L(F) \\
&= L(E+F)
\end{aligned}
$$

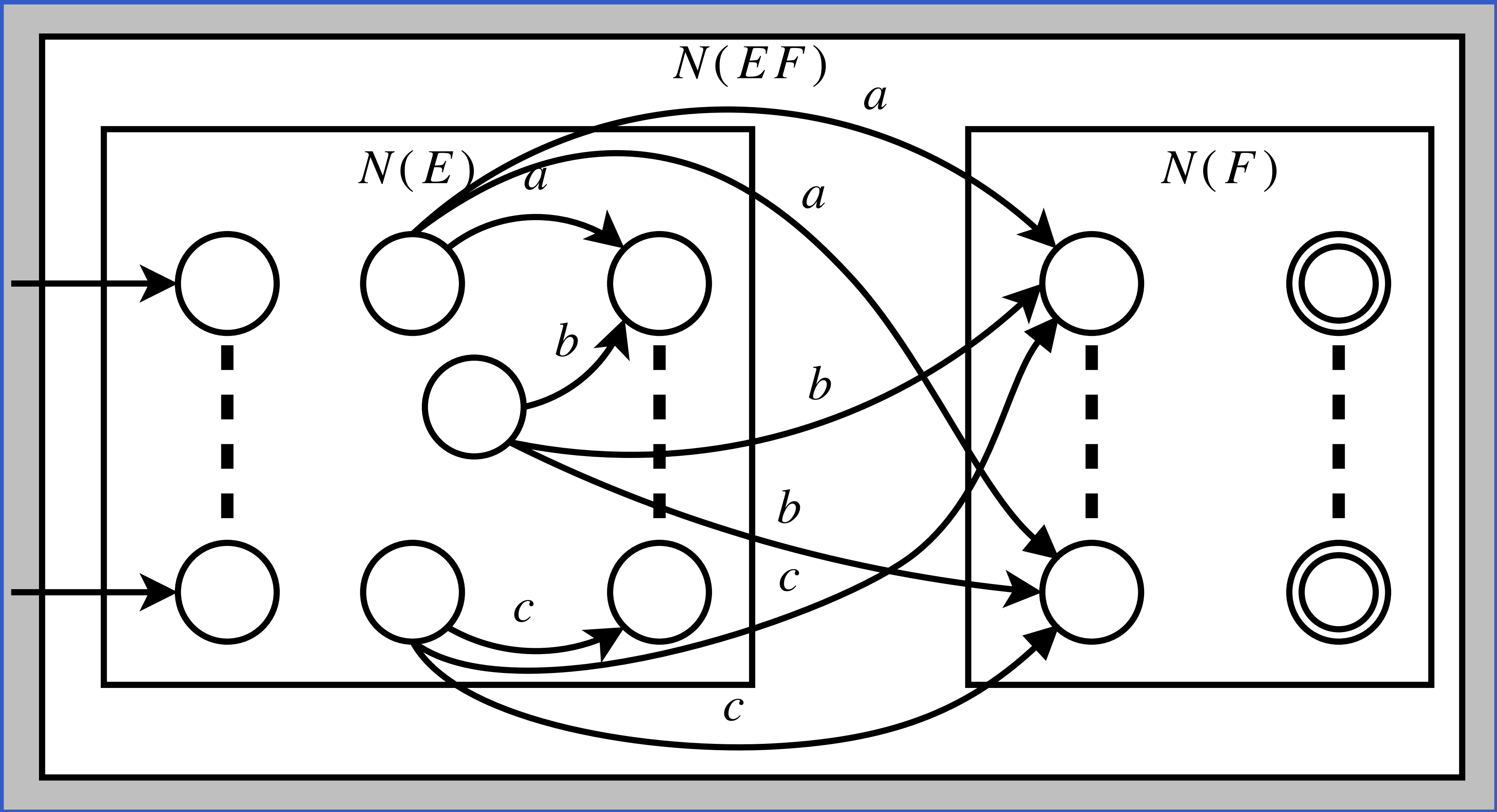
Thus, specification holds in this case.
(This is an ***inductive*** case.)

RE to NFA, Case EF (1)

Sub-case 1: No initial state of $N(E)$ is accepting; i.e. $\epsilon \notin L(N(E))$ (Recall: $L(EF) = L(E)L(F)$)

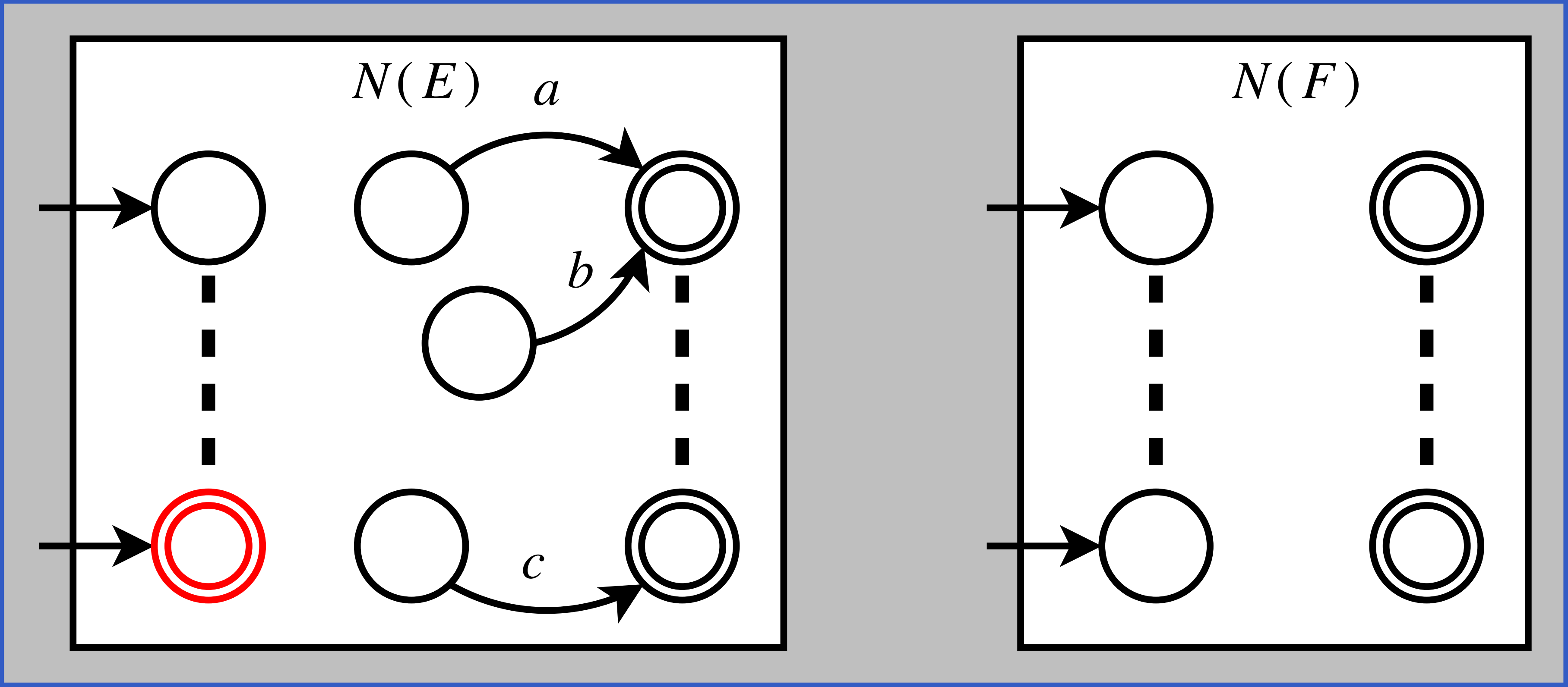


RE to NFA, Case EF (2)

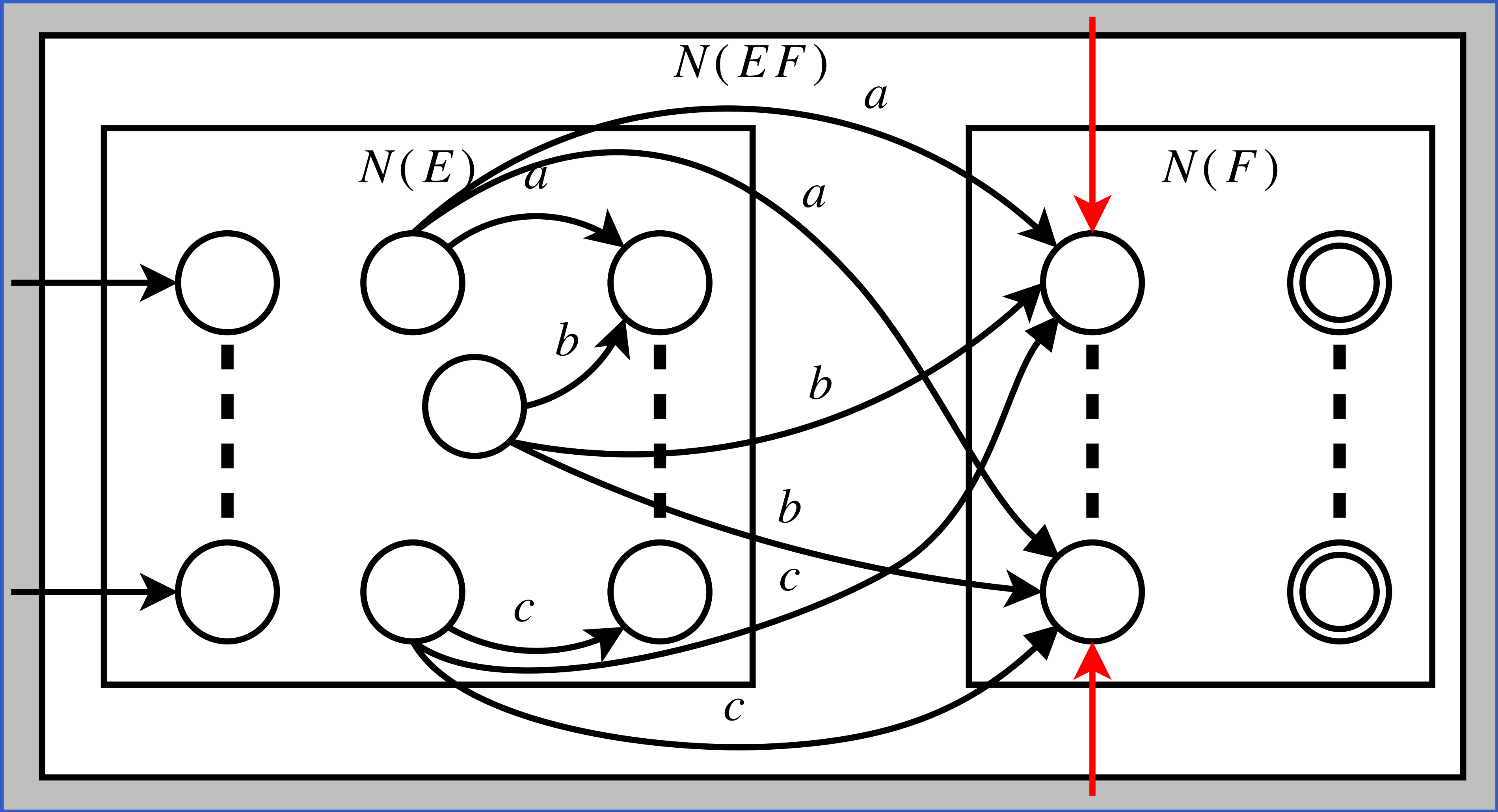


RE to NFA, Case EF (3)

Sub-case 2: Some initial states of $N(E)$ are accepting; i.e. $\epsilon \in L(N(E))$



RE to NFA, Case EF (4)



RE to NFA, Case EF (5)

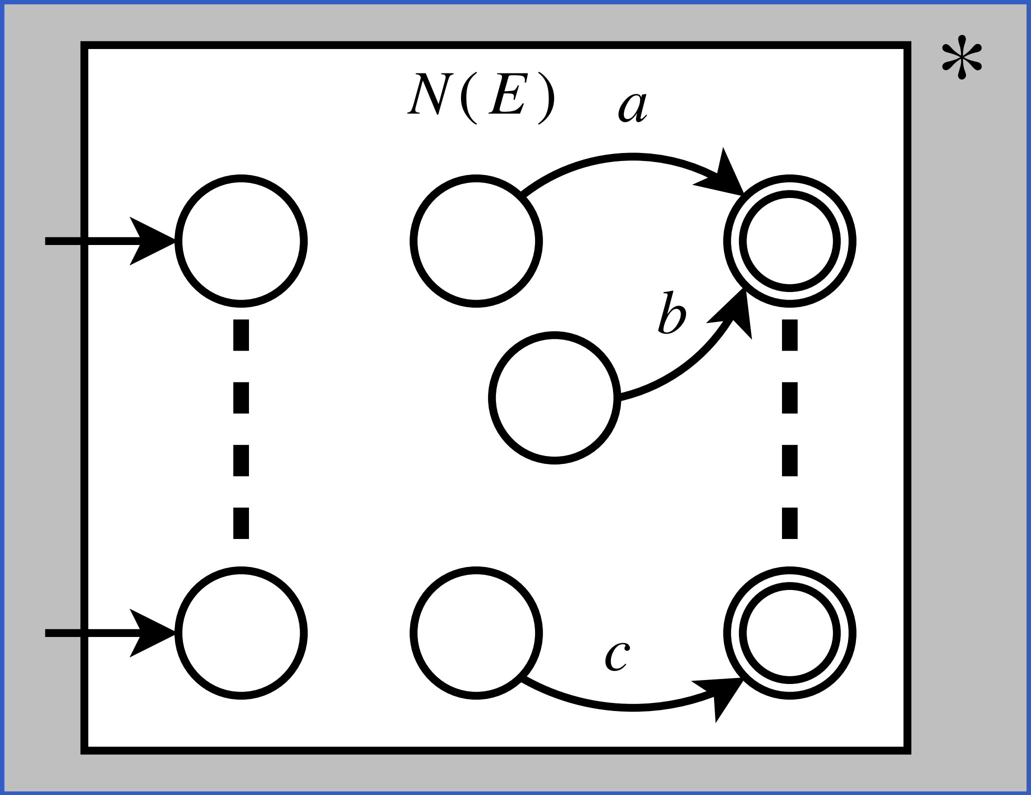
Note: Assuming specification holds for E and F,

$$\begin{aligned} L(N(EF)) &= L(N(E))L(N(F)) \\ &= L(E)L(F) \\ &= L(EF) \end{aligned}$$

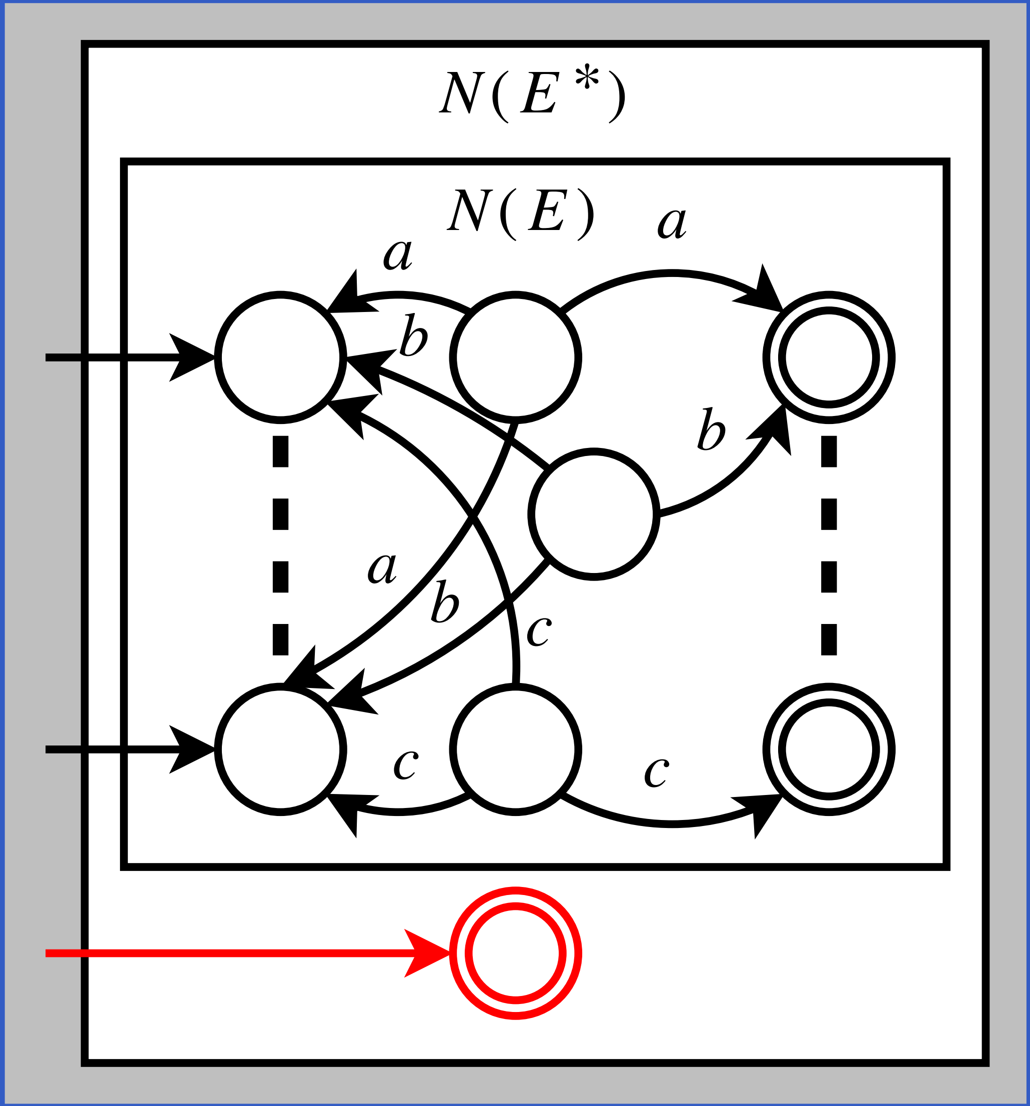
Thus, specification holds in this case.
(This is an ***inductive*** case.)

RE to NFA, Case E^* (1)

(Recall: $L(E^*) = L(E)^*$)



RE to NFA, Case E^* (2)



Note the additional initial and accepting state that ensures the empty word is accepted.

RE to NFA, Case E^* (3)

Note: Assuming specification holds for E,

$$\begin{aligned} L(N(E^*)) &= L(N(E))^* \\ &= L(E)^* \\ &= L(E^*) \end{aligned}$$

Thus, specification holds in this case.
(This is an ***inductive*** case.)

RE to NFA, Case (E)

(Recall: $L(\ (E)\ ) = L(E)$)

$N(\ (E)\ ) = N(E)$

Note: Assuming specification holds for E,

$$\begin{aligned} L(N(\ (E)\ )) &= L(N(E)) \\ &= L(E) \\ &= L(\ (E)\ ) \end{aligned}$$

Thus, specification holds in this case.
(This is an ***inductive*** case.)

Example

Systematically construct an NFA for the regular expression:

$$(\mathbf{a} + \mathbf{b})^*\mathbf{c}$$

(“zero or more as or bs, followed by a single c”)

Use the “graphical construction”. On the white board.

Scanning (1)

- The first stage of many real-world language processing tasks, such as a compiler, is to group individual characters into language-specific symbols called ***Lexemes*** or ***Tokens***:
  - Keywords (like **`if`**, **`then`**, **`while`**)
  - Literals (like **`42`**, **`3.14`**, **`'A'`**, **`"abc"`**)
  - Special symbols and separators (like **`:=`**, **`(`**, **`;`**)
  - ...

Scanning (1)

- The first stage of many real-world language processing tasks, such as a compiler, is to group individual characters into language-specific symbols called ***Lexemes*** or ***Tokens***:
  - Keywords (like **`if`**, **`then`**, **`while`**)
  - Literals (like **`42`**, **`3.14`**, **`'A'`**, **`"abc"`**)
  - Special symbols and separators (like **`:=`**, **`(`**, **`;`**)
  - . . .
- This process is called ***Lexical Analysis*** or ***Scanning***, and is performed by a ***Scanner***.

Scanning (2)

- Commonly, ***white space*** and ***comments*** are understood as ***token separators***.

Scanning (2)

- Commonly, ***white space*** and ***comments*** are understood as ***token separators***.
- An additional task of the scanner is often to ***discard*** white space and comments as they usually serve no purpose after the scanning.

Scanning (2)

- Commonly, ***white space*** and ***comments*** are understood as ***token separators***.
- An additional task of the scanner is often to ***discard*** white space and comments as they usually serve no purpose after the scanning.
- Regular expressions is the most commonly used formalism for describing the ***Lexical Syntax*** of a language; i.e. the syntax of the tokes, white space, and comments.

Scanning (2)

- Commonly, ***white space*** and ***comments*** are understood as ***token separators***.
- An additional task of the scanner is often to ***discard*** white space and comments as they usually serve no purpose after the scanning.
- Regular expressions is the most commonly used formalism for describing the ***Lexical Syntax*** of a language; i.e. the syntax of the tokes, white space, and comments.
- In essence, a scanner is thus a ***finite automaton***.

Scanning (3)

- There are many famous so called ***scanner generators***; e.g. Lex, Flex: given regular expressions describing the lexical syntax, they produce a scanner for the language.

Scanning (3)

- There are many famous so called ***scanner generators***; e.g. Lex, Flex: given regular expressions describing the lexical syntax, they produce a scanner for the language.
- In the following, we will study a ***hand-written scanner*** in Haskell for a simple language called ***TXL*** (for “Trivial eXpression Language”) to give a concrete example and practical experience of these ideas.

Scanning (3)

- There are many famous so called ***scanner generators***; e.g. Lex, Flex: given regular expressions describing the lexical syntax, they produce a scanner for the language.
- In the following, we will study a ***hand-written scanner*** in Haskell for a simple language called ***TXL*** (for “Trivial eXpression Language”) to give a concrete example and practical experience of these ideas.
- When studying the code, try to understand how ***the code actually implements a DFA***.

Lexical Syntax TXL (1)

$\texttt{'a'}$ etc. R.E. for ind. char.; Space etc. are “macros”.

$$
\begin{aligned}
\mathit{Space} &= \texttt{'\ '} + \texttt{'\textbackslash n'} \\
\mathit{Graphic} &= \texttt{'+'} + \texttt{'-'} + \texttt{'*'} + \texttt{'/'} \\
&\quad + \texttt{'('} + \texttt{')'} + \texttt{'='} \\
\mathit{Digit} &= \texttt{'0'} + \ldots + \texttt{'9'} \\
\mathit{Alpha} &= \texttt{'a'} + \ldots + \texttt{'z'} \\
\mathit{AlphaNum} &= \mathit{Alpha} + \mathit{Digit} \\
\mathit{LitInt} &= \mathit{Digit}\ \mathit{Digit}^{*} \\
\mathit{Id} &= \mathit{Alpha}\ (\mathit{Alpha} + \mathit{Digit})^{*} \\
\mathit{Keyword} &= \texttt{'l'}\ \texttt{'e'}\ \texttt{'t'} + \texttt{'i'}\ \texttt{'n'}
\end{aligned}
$$

Lexical Syntax TXL (2)

Finally, a regular expression for the entire language:

$$txl \quad = \quad (Graphic + LitInt + Id + Keyword + Space)^*$$

Ambiguity Issues (1)

The given regular expression accurately describes the lexical syntax of TXL, and is thus fine for ***checking*** if a string (word) belongs to TXL or not.

However, for the purpose of ***breaking a string into tokens***, it is not quite precise enough as there are ambiguities:

- Id and $Keyword$ ***overlaps***: is `let` an identifier or a keyword?
- Choice between ***long token or short tokens***: is `abc123` an identifier, an identifier `abc` and an integer literal `123`, or maybe even three identifiers followed by three literals?

Ambiguity Issues (2)

Such issues are commonly resolved by adopting certain conventions:

- ***Keywords takes precedence*** over identifiers; i.e., a token is an identifier only if it is not a keyword.
  (Thus, `let` is a keyword, not an identifier.)
- ***“Maximal Munch Rule”***: tokens should be as long as possible; i.e., prefer grouping as a single long token over a sequence of shorter ones.
  (Thus, `abc123` is a single token, an identifier.)

TXL Scanner (1)

```
type Id = String

data Token  =  T_Int Int
            |  T_Id Id
            |  T_Plus
            |  T_Minus
            |  T_Times
            |  T_Divide
            |  T_LeftPar
            |  T_RightPar
            |  T_Equal
            |  T_Let
            |  T_In
```

TXL Scanner (2)

```
lexer                 :: [Char] -> [Token]

-- End of input
lexer []              =  []

-- Drop white space and new lines
lexer (' '  : cs) =  lexer cs
lexer ('\n' : cs) =  lexer cs
```

TXL Scanner (3)

```
-- Lex simple tokens
lexer ('+' : cs) =  T_Plus     : lexer cs
lexer ('-' : cs) =  T_Minus    : lexer cs
lexer ('*' : cs) =  T_Times    : lexer cs
lexer ('/' : cs) =  T_Divide   : lexer cs
lexer ('(' : cs) =  T_LeftPar  : lexer cs
lexer (')' : cs) =  T_RightPar : lexer cs
lexer ('=' : cs) =  T_Equal    : lexer cs
```

TXL Scanner (4)

```
-- Lex literal integers, identifiers, and keywords
lexer (c : cs)
  | isDigit c = T_Int (read (c:takeWhile isDigit cs))
                : lexer (dropWhile isDigit cs)
  | isAlpha c = mkIdOrKwd (c:takeWhile isAlphaNum cs)
                : lexer (dropWhile isAlphaNum cs)
  | otherwise = error ("Unrecognised Character")
  where
    mkIdOrKwd         :: String -> Token
    mkIdOrKwd "let"   =  T_Let
    mkIdOrKwd "in"    =  T_In
    mkIdOrKwd cs      =  T_Id cs
```