50.051 Programming Language Concepts

W5-S2 Regular Expressions and Pattern Recognition in C

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Regular Expressions

Definition (Regular Expressions, or RegEx):

Regular Expressions (or Regex) are a powerful tool for pattern matching and searching in text.

They are used in many programming languages and applications to find, extract, and manipulate text based on specific patterns.

Regular expressions can be seen as a concise way of describing a set of strings that share a certain structure or patterns.

Later on, RegEx will be very useful for our compilers to recognize substrings of text that exhibit a certain pattern (e.g. they consist of a certain keyword, start/end with a certain character, etc.).

Regular Expressions

Definition (Regular Language):

Regular Language consists of the possible strings that exhibit the pattern defined by a given **Regular Expression**.

For instance, if the Regular Expression describes all the binary strings finishing with a 0…

…Then, all of these strings (100, 10100, 1110, 10, 1101010011010, etc.) will be considered **Regular Language** for the given **Regular Expression** we have defined above.

Technically, we have already played with RegEx FSMs! (Remember previous activities).

Can we use an FSM to implement a RegEx?

Yes!

Our Finite State Machines (FSM) from the previous lecture will be used to recognize regular languages, which are sets of strings that can be described by regular expressions.

There is a strong connection between regular expressions and FSMs: **for every regular expression, there exists an equivalent FSM that recognizes the same language. And vice versa.**

Technically, we have already played with RegEx FSMs! (Practice 1).

We would like to write an FSM with stopping states that will take strings *x* consisting of combinations of three characters: Z, A and M.

Possible combinations for the string *x* include, among many others, “ZAM”, “AMAZ”, “ZAMZAM”, etc.

Q1: Draw the FSM state diagram, where the FSM will accept an input string *x*, if and only if the string *x* contains the substring ZAM.

Q2 (cannot be solved for now, get back to it later): Could that be described using a RegEx of some sort?

The most basic RegEx

Definition (the most basic RegEx):

The most basic regex attempts to recognize **a string *x* containing a single letter, e.g. "a"**.

Any string that contains a (for instance, “a”, “aa”, “ba”, “singapore”, etc.) is then considered as regular language for this regular expression.

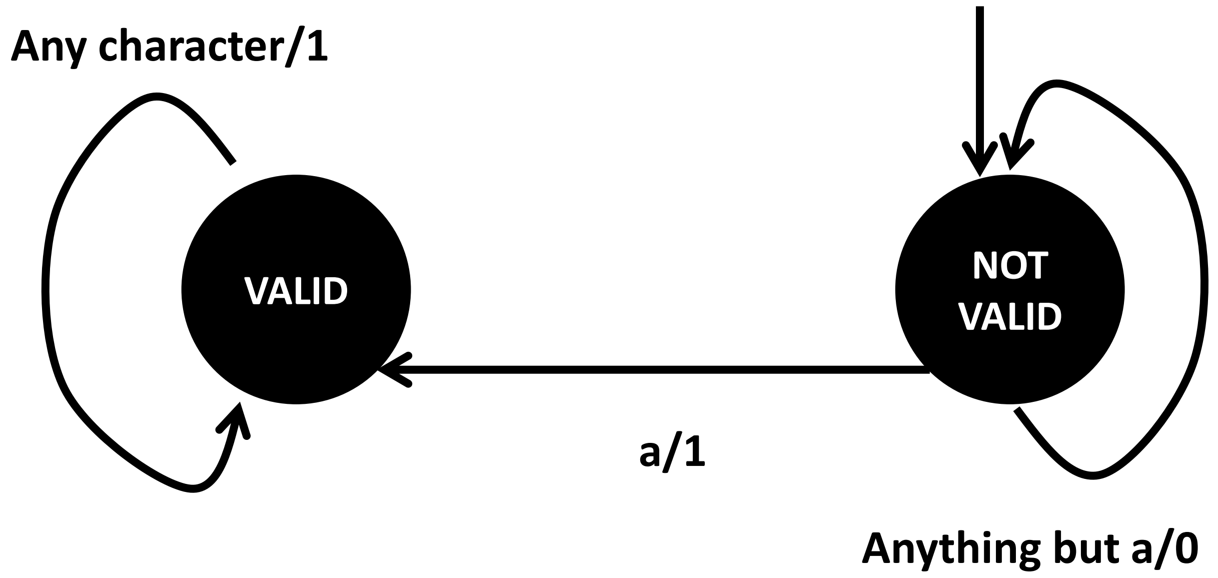
This regular expression is then simply denoted as “a”.

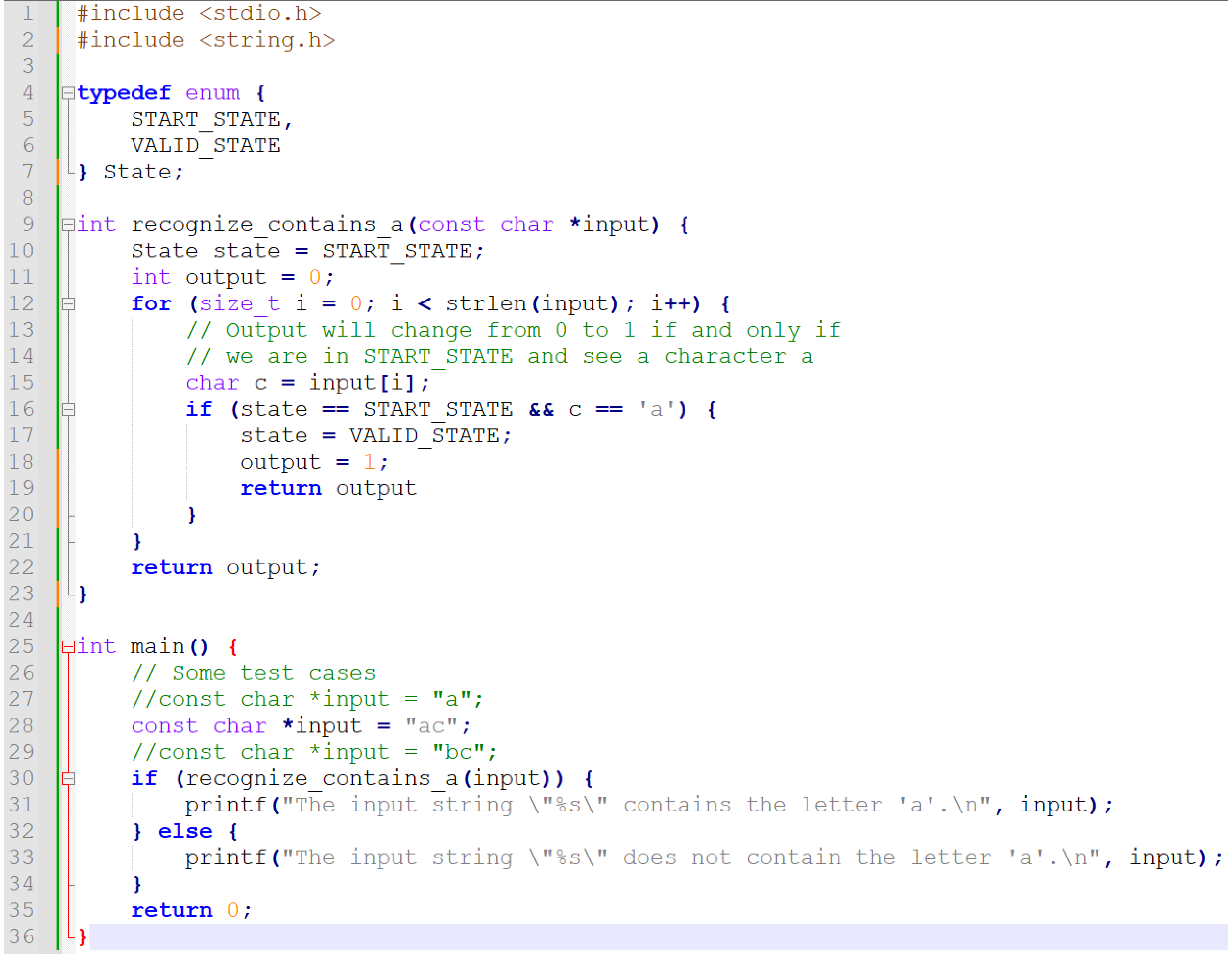
The most basic RegEx

This can be simply implemented using an FSM with outputs, as shown.

The FSM uses outputs and will stop early when an output 1 is produced.

The only acceptable input strings, will therefore be the ones that produce a 1 output at some point.





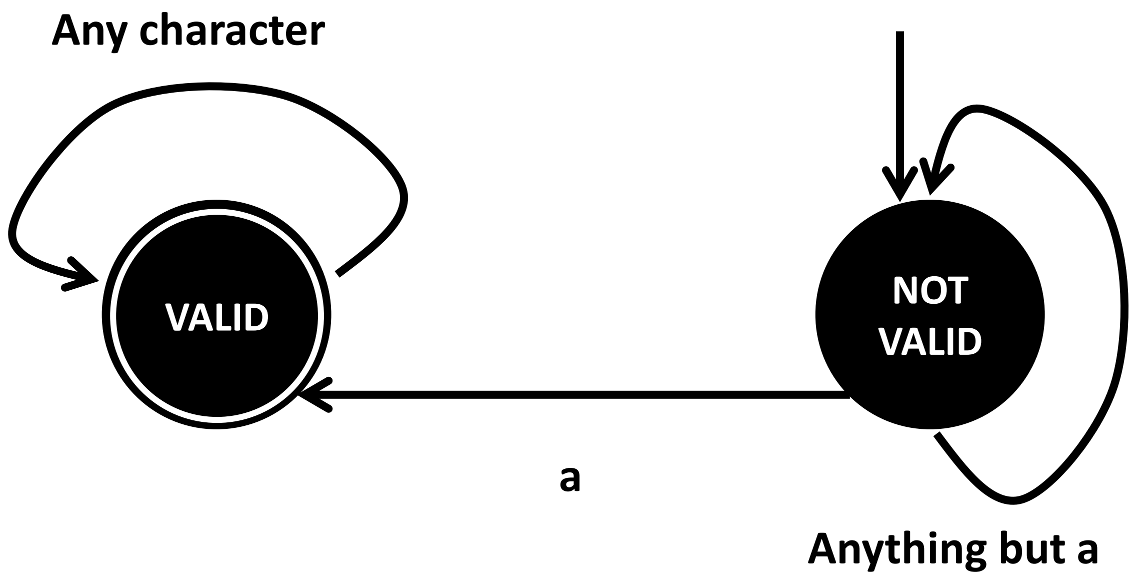
```c
#include <stdio.h>
#include <string.h>

typedef enum {
    START_STATE,
    VALID_STATE
} State;

int recognize_contains_a(const char *input) {
    State state = START_STATE;
    int output = 0;
    for (size_t i = 0; i < strlen(input); i++) {
        // Output will change from 0 to 1 if and only if
        // we are in START_STATE and see a character a
        char c = input[i];
        if (state == START_STATE && c == 'a') {
            state = VALID_STATE;
            output = 1;
            return output
        }
    }
    return output;
}

int main() {
    // Some test cases
    //const char *input = "a";
    const char *input = "ac";
    //const char *input = "bc";
    if (recognize_contains_a(input)) {
        printf("The input string \"%s\" contains the letter 'a'.\n", input);
    } else {
        printf("The input string \"%s\" does not contain the letter 'a'.\n", input);
    }
    return 0;
}
```

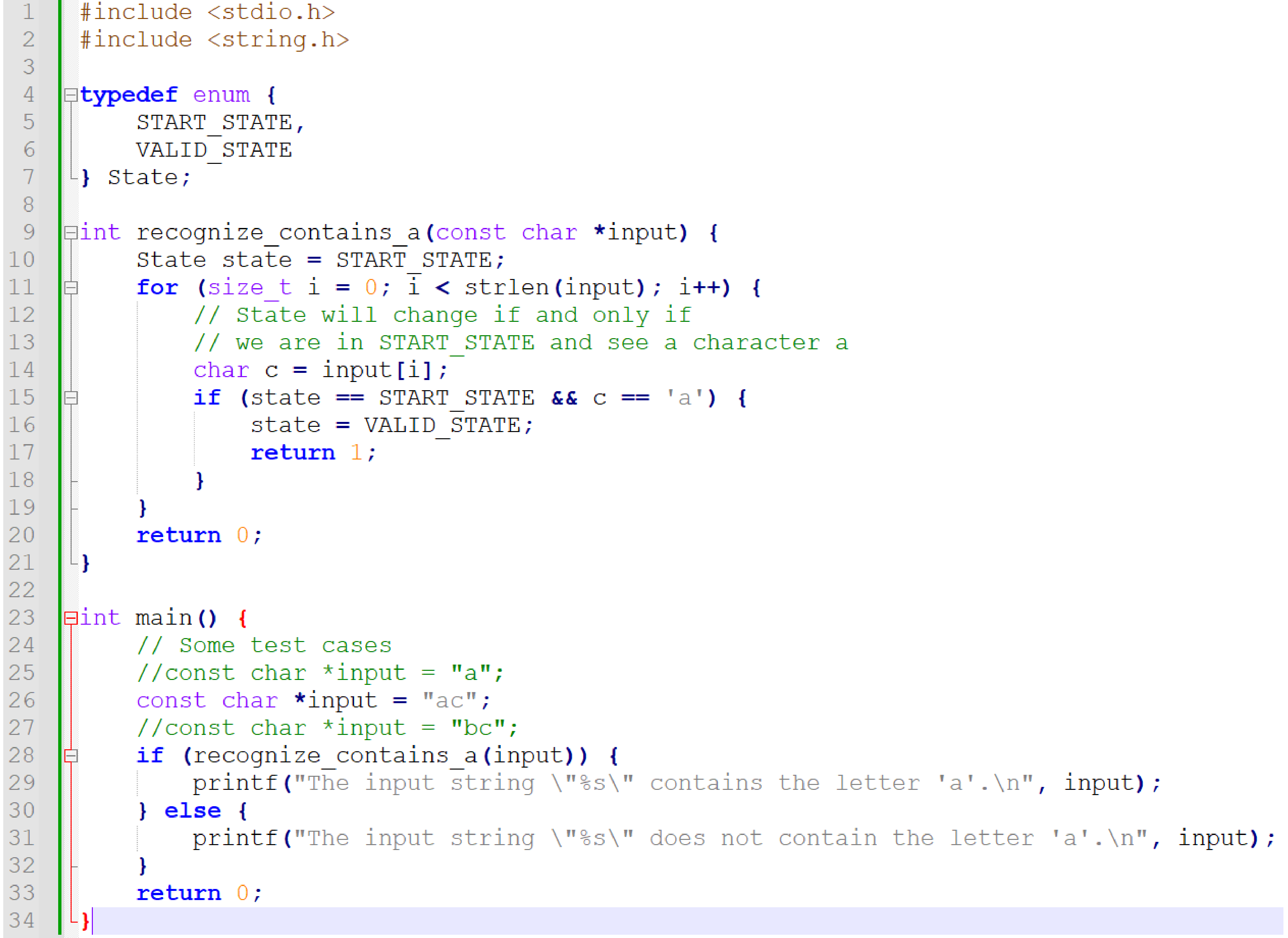
The most basic RegEx

This can also be implemented as an FSM with a single stopping state being VALID.

We can even decide to stop the FSM prematurely if the next state being produced at any given time is VALID.

This is roughly equivalent.





```c
#include <stdio.h>
#include <string.h>

typedef enum {
    START_STATE,
    VALID_STATE
} State;

int recognize_contains_a(const char *input) {
    State state = START_STATE;
    for (size_t i = 0; i < strlen(input); i++) {
        // State will change if and only if
        // we are in START_STATE and see a character a
        char c = input[i];
        if (state == START_STATE && c == 'a') {
            state = VALID_STATE;
            return 1;
        }
    }
    return 0;
}

int main() {
    // Some test cases
    //const char *input = "a";
    const char *input = "ac";
    //const char *input = "bc";
    if (recognize_contains_a(input)) {
        printf("The input string \"%s\" contains the letter 'a'.\n", input);
    } else {
        printf("The input string \"%s\" does not contain the letter 'a'.\n", input);
    }
    return 0;
}
```

The infamous Epsilon transition

Definition (the Epsilon transition):

In (Non-deterministic) Finite State Machines, the **Epsilon transition** is an optional transition which offers to advance to a next state, without consuming any character from the input string *s*.

In Layman terms, an epsilon transition literally means "you may choose to go to the next state (or not) without using the next input character yet".

Important note: When using an **Epsilon transition**, you do NOT skip the current input character. It is simply a free transition you may or may not decide to use when given the opportunity.

About the Non-Deterministic FSMs

Definition (Non-Deterministic Finite State Machines):

The Epsilon transitions are optional, and you may or may not decide to take them at a given time.

This basically means that for a given input string *x*, **there might be multiple possible paths in the FSM that can be taken**.

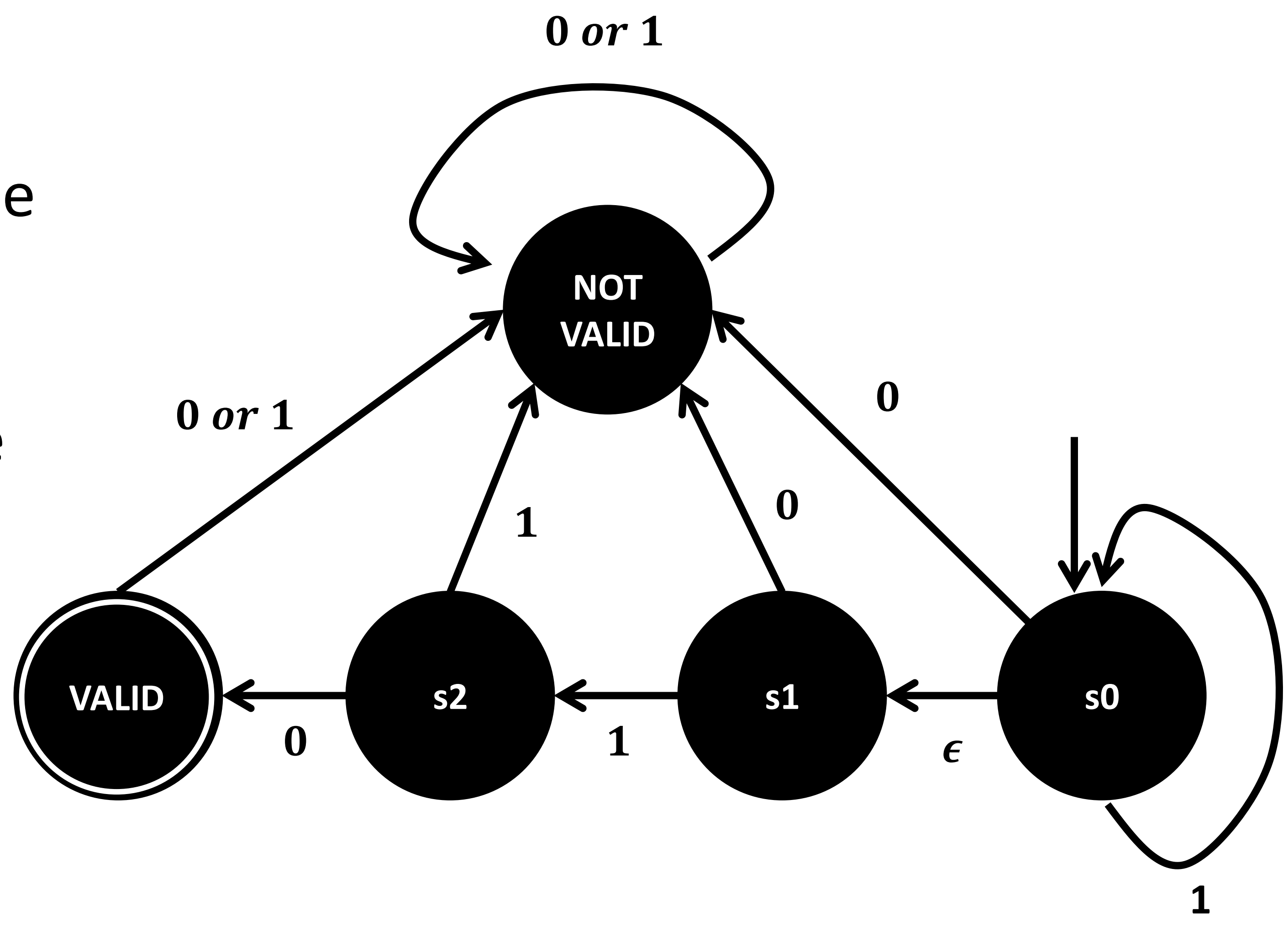
These types of FSM are commonly referred to as Non-Deterministic Finite State Machines, or NDFSM.

A given NDFSM will then consider a string *x* to be **acceptable** if and only if **there exists at least one path in the FSM that satisfies conditions in terms of stopping states or valid outputs being produced**.

A use case for Epsilon (Practice 2)

As an example of the use for the Epsilon transition, consider the NDFSM on the right.

Question: What are acceptable inputs for this NDFSM?

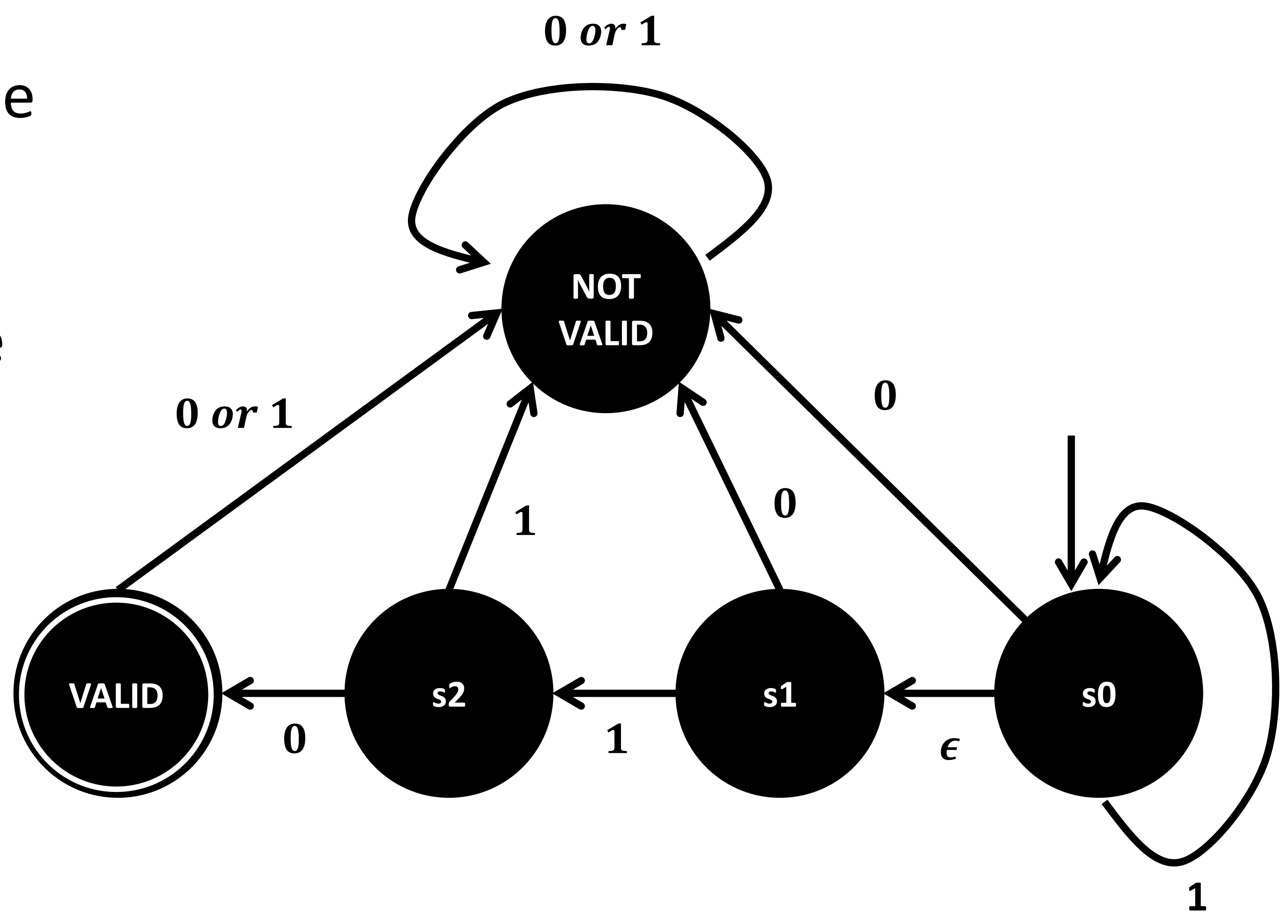


A use case for Epsilon (Practice 2)

As an example of the use for the Epsilon transition, consider the NDFSM on the right.

Question: What are acceptable inputs for this NDFSM?

Answer: Any string that consists of any number of ones (but at least one), and then finishes with exactly one zero (e.g. 10, 110, 1110, 111111110, etc.)



On the use of the Epsilon Transitions in RegEx

Definition (Epsilon transitions in RegEx):

One important use of epsilon transitions is in the construction of NDFSMs for RegEx.

In this context, an epsilon transition can typically be used to **represent the empty string**, which is often considered a valid input for many regular expressions.

By including epsilon transitions in the FSM of your RegEx,

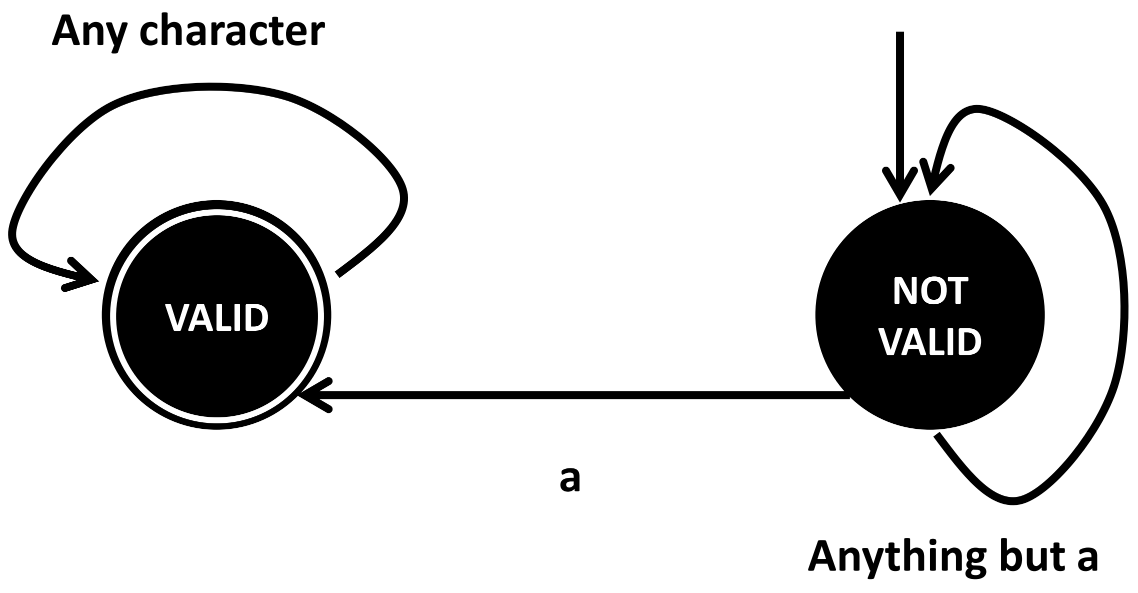
- the regular expression can be represented more simply,
- and you may use it to combine several RegEx into a single combined one (as will be shown next).

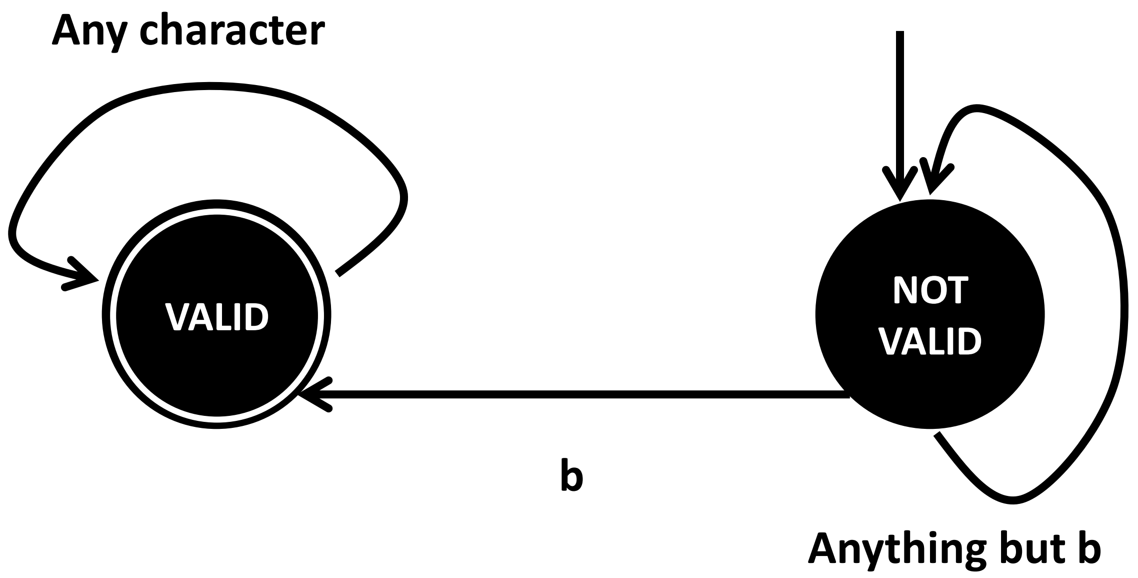
Concatenation of RegEx

Theorem (Concatenation of RegEx):

Assume that we have two RegEx, e.g.

- our first RegEx is "a"
- and our second RegEx is "b",
- and we have defined their respective two FSMs.





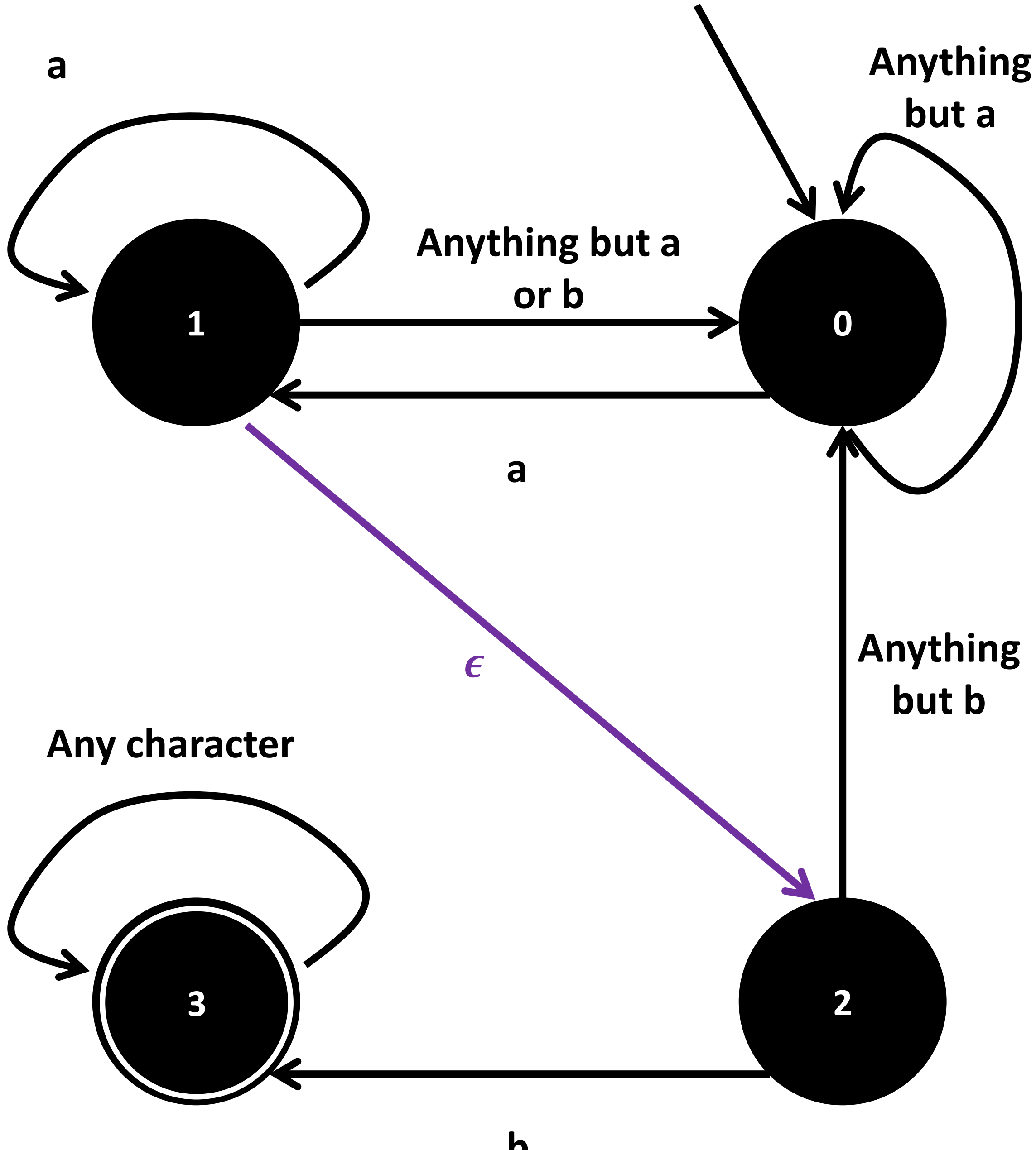
Concatenation of RegEx

Theorem (Concatenation of RegEx):

Assume that we have two RegEx, e.g.

- our first RegEx is “a”
- and our second RegEx is “b”,
- and we have defined their respective two FSMs.

The **concatenated RegEx “ab”**, which accepts strings containing the substring “ab”, can then be built **by combining the two FSMs, connecting them with an Epsilon transition.**



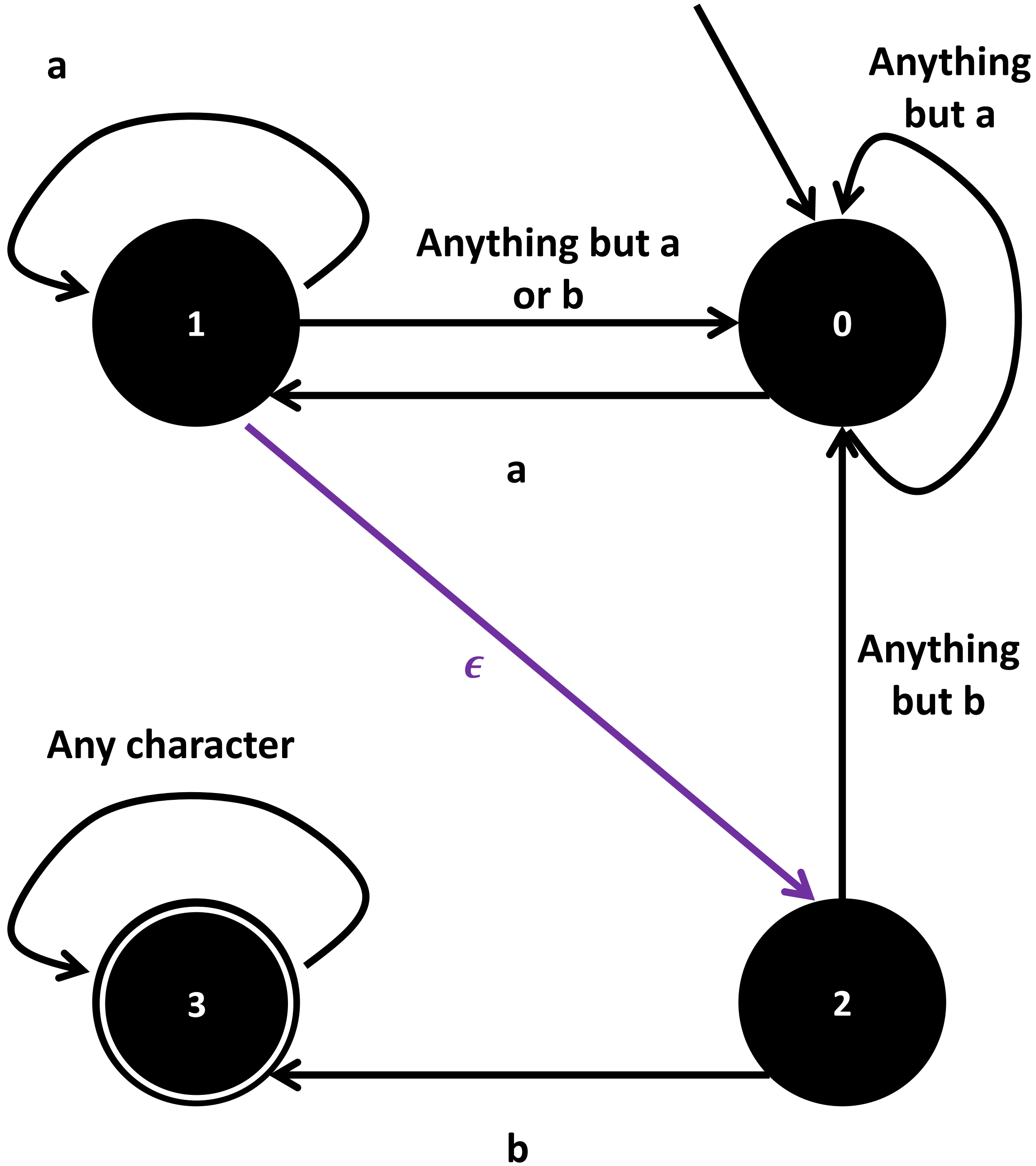
Concatenation of RegEx

Question: Could we transform our NDFSM into a DFSM if we

- merge both states 1 and 2 together,
- to get rid of the Epsilon transition here,

And still obtain the expected RegEx that we want?

(**Theorem:** Any NDFSM can be transformed into a DFSM.)



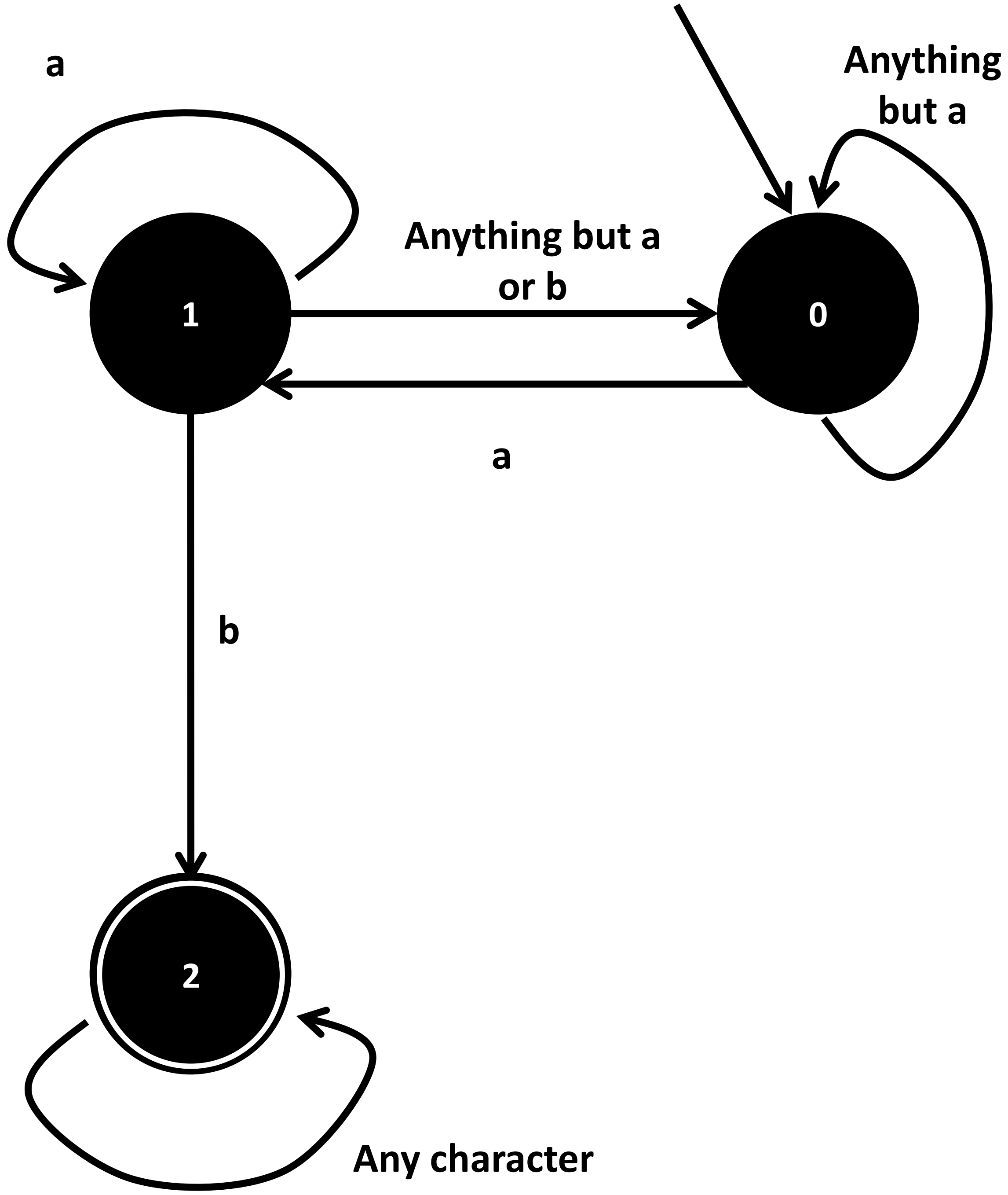
Concatenation of RegEx

Question: Could we transform our NDFSM into a DFSM if we

- merge both states 1 and 2 together,
- to get rid of the Epsilon transition here,

And still obtain the expected RegEx that we want?

Would that be valid?



Back to our Practice 1

We would like to write an FSM with stopping states that will take strings *x* consisting of combinations of three characters: Z, A and M.

Possible combinations for the string *x* include, among many others, “ZAM”, “AMAZ”, “ZAMZAM”, etc.

Draw a FSM state diagram, which has the FSM produce a specific output, <u>if and only if the string *x* contains the substring ZAM.</u>

Q2 (can probably be solved now): Could that be described using a RegEx of some sort?

Choice of RegEx

Theorem (Choice operator in RegEx):

Assume that we have two RegEx, e.g.

- our first RegEx is "a"
- and our second RegEx is "b",
- and we have defined their respective two FSMs.

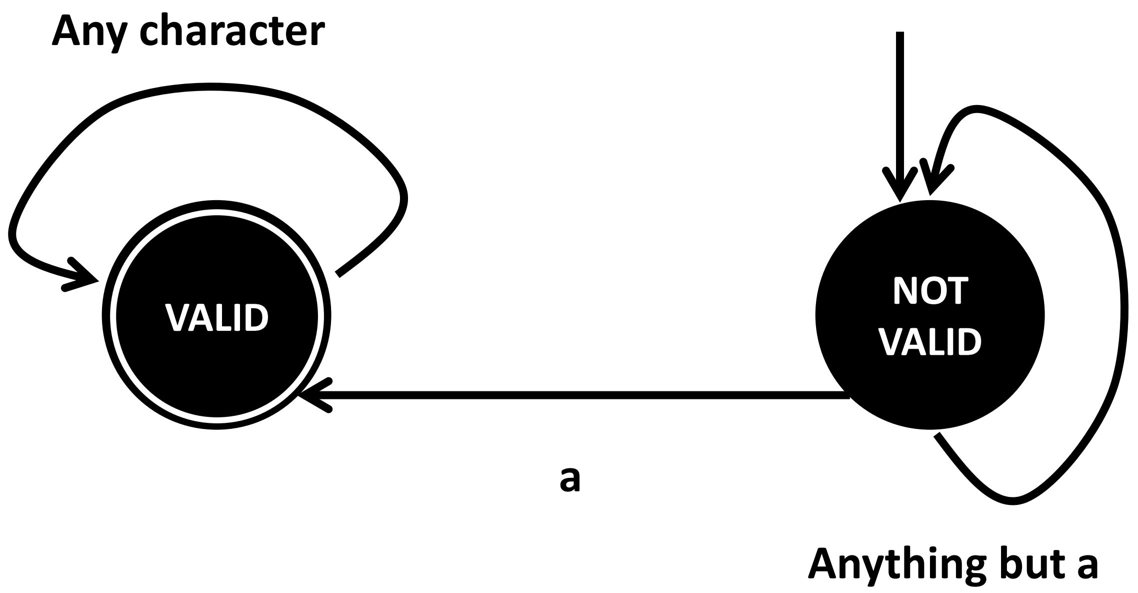
The choice **operator |** between both **RegEx "a|b"** is used to define the union of two regular language consisting of the two strings "a" and "b".

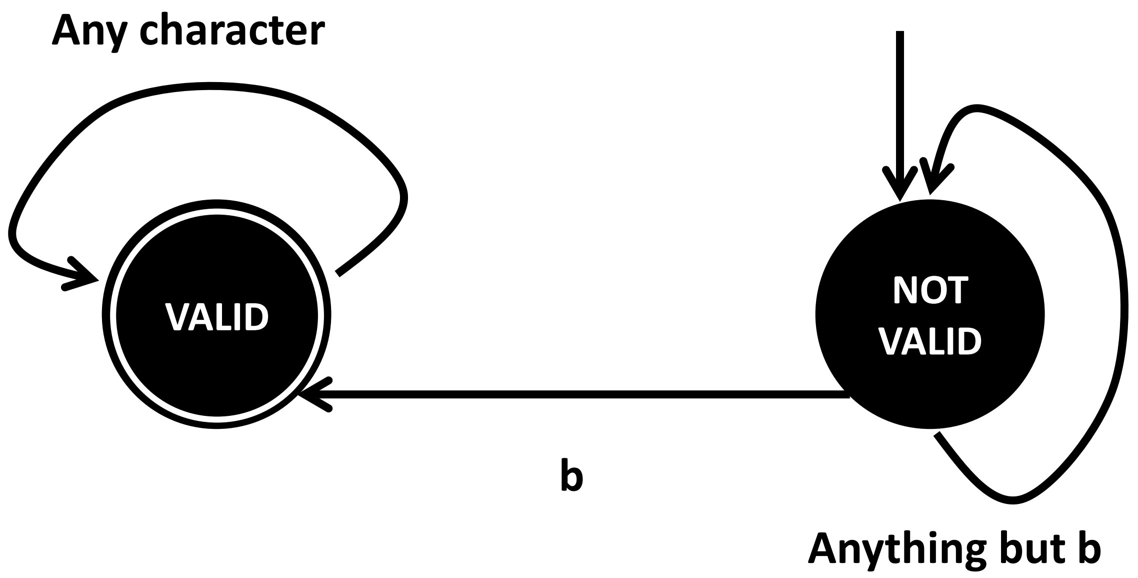
Its regular language consists of the union set of strings that the letter a, strings that contain the b, and strings that contain both.

Choice of RegEx

As before it is possible to build the FSM for “a|b” by

- combining the two FSMs for “a” and “b”,
- in **parallel**, this time,
- connecting them with Epsilon transitions.



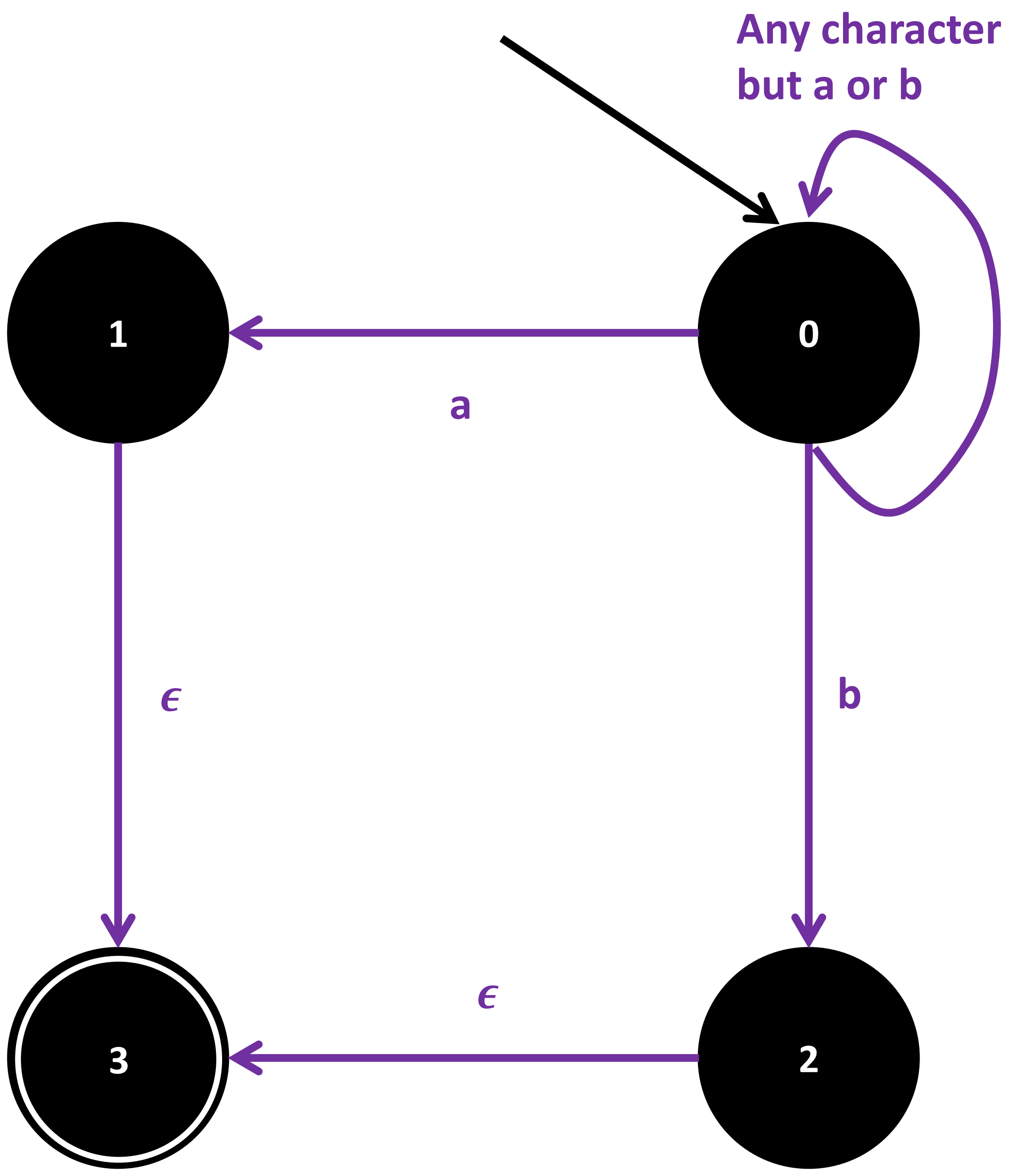


Choice of RegEx

As before it is possible to build the FSM for “a|b” by

- combining the two FSMs for “a” and “b”,
- in **parallel**, this time,
- connecting them with Epsilon transitions.

(Note: Each state 1, 2, 3 has self-links matching any character, removed for simplicity.)

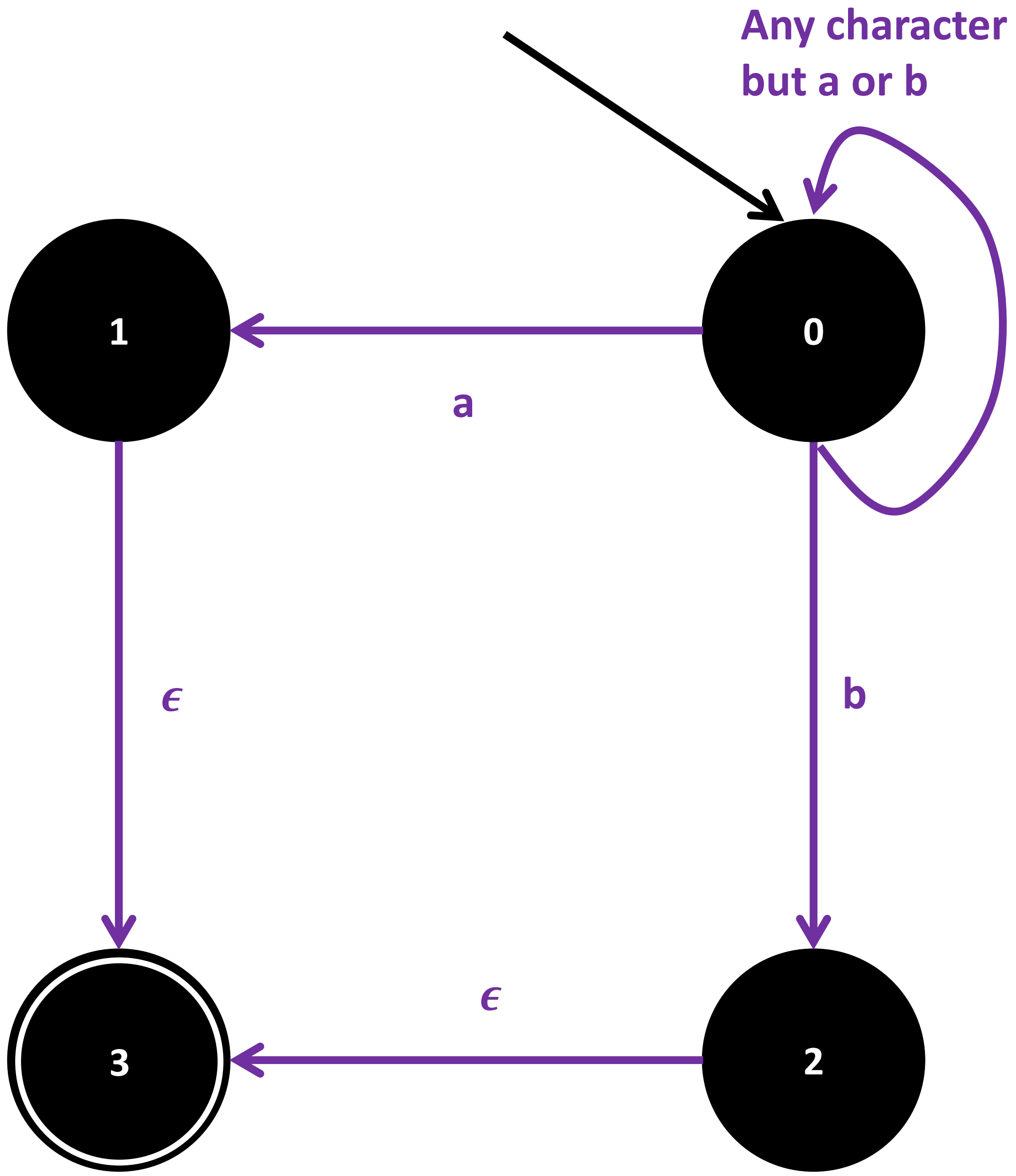


Choice of RegEx

Quick question: Again, could we transform our NDFSM into a DFSM if we decide to

- merge states 1, 2 and 3 together,
- to get rid of the Epsilon transitions here,

And still obtain the expected RegEx that we want?



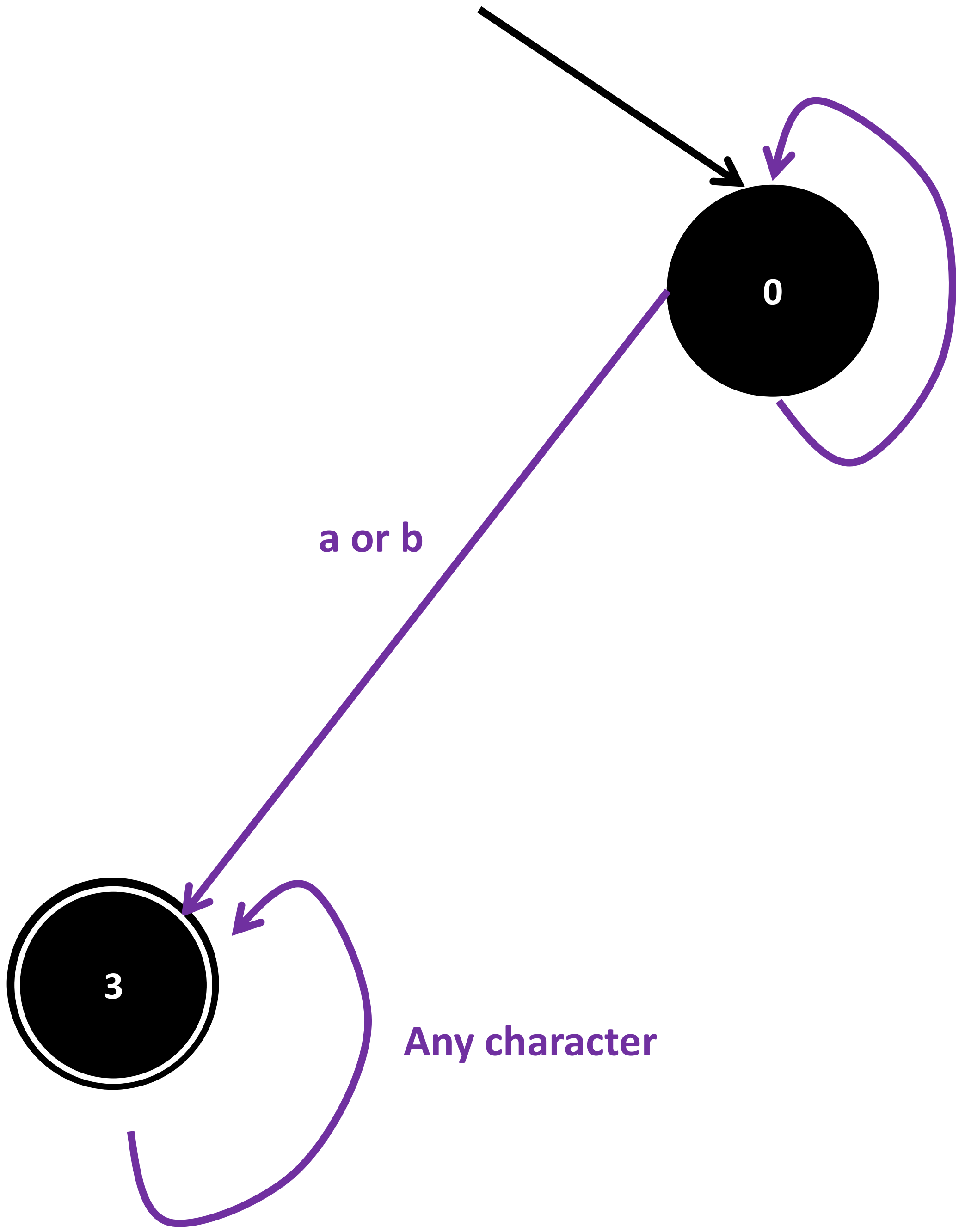
Choice of RegEx

Quick question: Again, could we transform our NDFSM into a DFSM if we decide to

- merge states 1, 2 and 3 together,
- to get rid of the Epsilon transitions here,

And still obtain the expected RegEx that we want?

Would this work?



On the use of parentheses in RegEx

Definition (On the use of parentheses in RegEx):

Parentheses can be used in RegEx to separate operations.

For instance, we could write the RegEx **“(a|b)c”**, whose acceptable language consists then of all the substrings containing either one of the two strings **“ac”** and **“bc”**.

It should not be confused with **“a|bc”**, whose regular language is then consisting of all the substrings containing either one of the two strings “a” and “bc”.

In this second case, **a|bc is equivalent to a|(bc).**

On the use of the escape character in RegEx

Definition (On the use of the escape character in RegEx):

The **Escape character \** can be used in RegEx to indicate that a special symbol (or something we call a metacharacter in RegEx) – which normally has an effect in RegEx – should in fact appear in the string.

For instance, we could write the RegEx **“\\(a\\)c”**, whose regular language consists then of all the strings containing the substring **“(a)c”.**

In this case, the parentheses symbols now being treated as characters to appear in a given input string *x*.

Closure or Kleene operator

Theorem (Closure or Kleene operator in RegEx):

Assume that we have a RegEx “a” and its respective FSM.

The **Closure (or Kleene) operator *** is used to define the RegEx **“a*”**.

Its regular language consists of **all the strings containing zero or more repetitions of the character a**, for instance “”, “a”, “aa”, “aaa”, “aaaaaaaaaaaaaaaaaaaaa”, etc.

Closure or Kleene operator

Theorem (Closure or Kleene operator in RegEx):

Assume that we have a RegEx “a” and its respective FSM.

The **Closure (or Kleene) operator *** is used to define the RegEx **“a*”**.

Its regular language consists of **all the strings containing zero or more repetitions of the character a**, for instance “”, “a”, “aa”, “aaa”, “aaaaaaaaaaaaaaaaaaaaa”, etc.

Crazy Question: Does that mean that “c” would match “a*”, as it contains zero repetitions of the character “a”?!

An important note: partial vs full matching

It is important to note that, we often consider **partial matching (or substring matching)** when our RegEx consists of basic operation combined with concatenation and/or choice operations.

Once a Kleene operator appears, it is often preferable to consider **full string matching** instead**.**

If **full string matching** is considered, then the RegEx **“a*” would only match an empty string or strings that are composed exclusively of the letter 'a'**, such as "a", "aa", "aaa", etc.

It would not match "aaaab" or "baaa" because these strings contain additional characters beyond just 'a'.

Full string matching vs. substring matching

Definition (Full String Matching):

Full string matching, also known as exact matching, is when the RegEx is used to match an entire string from start to finish without any extra characters before or after the pattern.

This means that the entire string must conform exactly to the pattern described by the RegEx for a match to be considered part of its regular language.

*If **full string matching** is considered, then **it would only match an empty string or strings that are composed exclusively of the letter 'a'**, such as "a", "aa", "aaa", etc. It would not match "aaaab" or "baaa" because these strings contain additional characters beyond just 'a'.*

Full string matching vs. substring matching

Definition (Partial String Matching):

Substring matching, on the other hand, involves checking if there are at least one occurrence of the pattern within a larger string.

The RegEx engine searches through the string and matches any part of it that conforms to the regex pattern, regardless of what characters may be present before or after the match.

*If **partial matching (or substring matching)** is considered, then **"a*" would match any other characters than the sequence of "a" characters**. This means that "aaaab" or "caa" would be accepted as regular language for "a*".*

Closure or Kleene operator

Theorem (Closure or Kleene operator in RegEx):

Assume that we have a RegEx “a” and its respective FSM.

The **Closure (or Kleene) operator *** is used to define the RegEx **“a*”**.

Its regular language consists of **all the strings containing zero or more repetitions of the character a**, for instance “”, “a”, “aa”, “aaa”, “aaaaaaaaaaaaaaaaaaaaaa”, etc.

Crazy Question: Does that mean that “c” would match “a*”, as it contains zero repetitions of the character “a”?!

→ **So how to control if full string matching or partial matching then?**

Beginning (^) and End ($) of a string

In RegEx, the caret (^) and dollar sign ($) are both metacharacters that represent the **beginning** and **end** of a string, respectively.

- **^**: The caret (^) denotes the beginning of a string.
- **$**: The dollar sign ($) denotes the end of a string.

E.g., the regex ^a (resp. a$) matches any string that starts (resp. ends) with "a".

For instance,

- The regex ^abc$ accepts only the string "abc", but not "aabc", "abcc", or “abcabc".
- The regex ^abc checks for "abc" at the beginning of a string.
  It accepts "abcd" but will not accept "aabc".
- The regex abc$ matches "abc" at the end of a string. Same idea as above.

Full string matching vs. substring matching

In the context of regex, **full string matching is often implemented by anchoring the regex pattern to the beginning and end of the string using the caret (^) and dollar sign ($) respectively**.

For example, the regex pattern **^a*$** would match an empty string or any string that is composed exclusively of the letter 'a', such as "a", "aa", "aaa", etc., but it would not match "aaaab" or "baaa" because these st
On the other hand, the regex pattern **a*** without anchors would implement **partial matching**. It would find matches within any string that contains the letter 'a'. In "aaaab", it would match "aaaa", and in "caa", it would match "aa".

Closure or Kleene operator

Theorem (Closure or Kleene operator in RegEx):

Assume that we have a RegEx “a” and its respective FSM.

The **Closure (or Kleene) operator *** is used to define the RegEx **“a*”**.

Its regular language consists of **all the strings containing zero or more repetitions of the character a**, for instance “”, “a”, “aa”, “aaa”, “aaaaaaaaaaaaaaaaaaaaa”, etc.

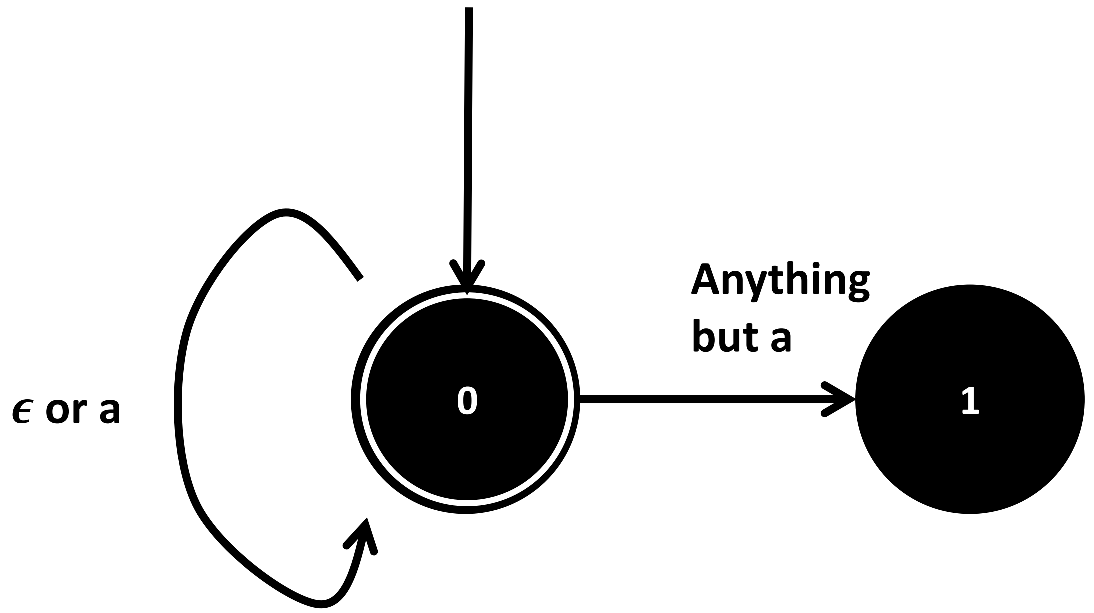
Using the Regex **“a*”** suggests that we use **partial matching**.

On the other hand, using **“^a*$”** suggests a **full string matching**.

Closure or Kleene operator

The FSM for “^a*$”, in the case of full string matching, is again, based on the one for “a” and is shown.

In that case, the use of epsilon is necessary to match the empty string as a possible regular language for “a*”.



Variations of the Kleene operator

Theorem (Plus or Kleene Plus operator in RegEx):

Assume that we have a RegEx “a” and its respective FSM.

The **Plus (or Kleene Plus) operator +** is used to define the RegEx **“a+”**.

Its regular language, in **full string matching**, consists of **all the strings containing of one or more repetitions of the character a**, i.e. “a”, “aa”, “aaa”, “aaaaa”, etc.

The RegEx **“a+”** is a shorter notation equivalent to **“aa*”**.

Variations of the Kleene operator

Theorem (Question Mark or Optional operator in RegEx):

Assume that we have a RegEx “a” and its respective FSM.

The **Question Mark (or Optional operator) ?** is used to define the RegEx **“a?”**.

Its regular language, in **full string matching**, consists of **all the strings consisting of exactly zero or one repetitions of the character a**, i.e. “” and “a” only.

The RegEx **“a?”** is a shorter notation equivalent to **“a|ϵ”**, where ϵ **denotes the empty string**.

The Kleene Theorem

Theorem (The Kleene Theorem):

The three fundamental operators we have defined earlier:

- Concatenation “ab”,
- Choice “a|b”
- And Kleene “a*”,
- (Along with the Epsilon symbol,)

can be used to define every other operation in RegEx.

For instance,

- “a+” is equivalent to “aa*”,
- “a?” is equivalent to “a|ϵ”,
- Etc.

Also, keep in mind that the **operator precedence** in RegEx is then defined as:

$$(R) > R^* > R_1R_2 > R_1|R_2$$

Quiz Time!

Which regular expression would be the best to match the two strings "cat" and "bat"?

A. ^(c|b)at$

B. ^(c|b)?at$

C. ^(c+b)at$

D. ^(c|b)+(at)$

Quiz Time!

Which regular expression would be the best to match the two strings "cat" and "bat"?

A. ^(c|b)at$

B. ^(c|b)?at$

C. ^(c+b)at$

D. ^(c|b)+at$

(Some others could work but they are terrible malpractice, so not going to accept them)

Quiz Time!

What does the Kleene star (*) operator represent in regular expressions?

A. Match exactly one character

B. Match zero or more repetitions of a character

C. Match one or more repetitions of a character

D. Match any single character

Quiz Time!

What does the Kleene star (*) operator represent in regular expressions?

A. Match exactly one character

B. Match zero or more repetitions of a character

C. Match one or more repetitions of a character

D. Match any single character

Quiz Time!

Which regular expression would match the strings "ab", "abb", "abbb", and so on, but no other strings? The string “a” is not acceptable.

A. ^abb*$

B. ^ab$

C. ^ab+$

D. ^a+ab$

Quiz Time!

Which regular expression would match the strings "ab", "abb", "abbb", and so on, but no other strings? The string “a” is not acceptable.

A. **^abb*$ (but could be written in a simpler manner)**

B. ^ab$

C. **^ab+$ (equivalent to aab* as b+ is equivalent to bb*)**

D. ^a+ab$

Quiz Time!

Which of the strings below is considered regular language for the regular expression “a\?b” ?

A. a

B. ab

C. a?b

D. a\b

E. ca?bddd

Quiz Time!

Which of the strings below is considered regular language for the regular expression “a\?b” ?

A. a

B. ab

C. a?b

D. a\b

E. ca?bddd

The Square Bracket notation in RegEx

Definition (The Square Bracket notation in RegEx):

In RegEx, the square brackets [] denote a character class, which is a set of characters that can match a single character in our string *x*. For instance,

- [a-z]: matches any lowercase letter from a to z,
- [A-Z]: matches any uppercase letter from A to Z,
- [0-9]: matches any digit.

This [0-9] notation is a convenient replacement for its equivalent (0|1|2|3|4|5|6|7|8|9).

Negating a grouping

A quick note on the Caret symbol ^:

When used inside a character class (i.e., between square brackets), the caret symbol **^ negates the character class**.

For example,

- the regex ^a matches any string that begins with the character "a",
- while the regex [^a] matches any character that is not "a".

Additional notations

More groupings can be used in RegEx

- **.** : The dot (.) is a metacharacter that matches any single character except for a newline character. For example, the regular expression "a.b" would match "axb", "a7b", "a b", and so on, but not "a\nb".
- **\d**: This character class matches any digit character (i.e., 0-9).
- **\w**: This character class matches any "word" character, which includes uppercase and lowercase letters, digits, and underscore symbol.
  It is equivalent to ([a-zA-Z0-9_].

Additional notations

More groupings can be used in RegEx

- **\s**: This character class matches any whitespace character, including spaces, tabs, and newline characters.
- **[\t\n\r\f\v]**: This character class matches any whitespace character, including tabs (\t), newline characters (\n), carriage returns (\r), form feeds (\f), and vertical tabs (\v). It's equivalent to \s, but it doesn't include spaces.

Quiz time!

Consider the regular expression “^[A-Z]+$”, which strings below are considered regular language for this RegEx?

A. “” (empty string)

B. “B”

C. “ABC”

D. “A1B2C3”

E. “a”

F. “abC”

G. “123”

Quiz time!

Consider the regular expression “^[A-Z]+$”, which strings below are considered regular language for this RegEx?

A. “” (empty string)

B. “B”

C. “ABC”

D. “A1B2C3”

E. “a”

F. “abC”

G. “123”

Quiz time!

What does the regular expression \w+ match?

A. Any string that contains at least one whitespace character
B. Any string that starts with a digit and ends with a letter
C. Any string that contains one or more “word” characters
D. Any string that contains exactly one “word” character

Quiz time!

What does the regular expression \w+ match?

A. Any string that contains at least one whitespace character
B. Any string that starts with a digit and ends with a letter
C. Any string that contains one or more “word” characters
D. Any string that contains exactly one “word” character

Quiz time!

What does the regular expression \scat\s match?

A. Any string that contains the substring “cat“

B. Any string that starts with the letter “c” and ends with the letter “t”

C. Any string that contains the substring “cat” surrounded by whitespace characters

D. Any string that contains the substring “cat” surrounded by any characters

Quiz time!

What does the regular expression \scat\s match?

A. Any string that contains the substring “cat“
B. Any string that starts with the letter “c” and ends with the letter “t”
C. Any string that contains the substring “cat” surrounded by whitespace characters
D. Any string that contains the substring “cat” surrounded by any characters

Quiz time!

What does the regular expression ^[a-z]+\d?$ match?

A. Any string that starts with a lowercase letter and ends with a digit
B. Any string that starts with a digit and ends with a lowercase letter
C. Any string that starts and uses one or more lowercase letters, and optionally ends with a digit
D. Any string that contains exactly one lowercase letter followed by a digit

Quiz time!

What does the regular expression ^[a-z]+\d?$ match?

A. Any string that starts with a lowercase letter and ends with a digit

B. Any string that starts with a digit and ends with a lowercase letter

C. Any string that starts and uses one or more lowercase letters, and optionally ends with a digit

D. Any string that contains exactly one lowercase letter followed by a digit

The RegEx library in C

The RegEx library is a library of functions and tools for working with regular expressions in C (among other languages), and provides a set of functions and tools for compiling, matching, and manipulating regular expressions

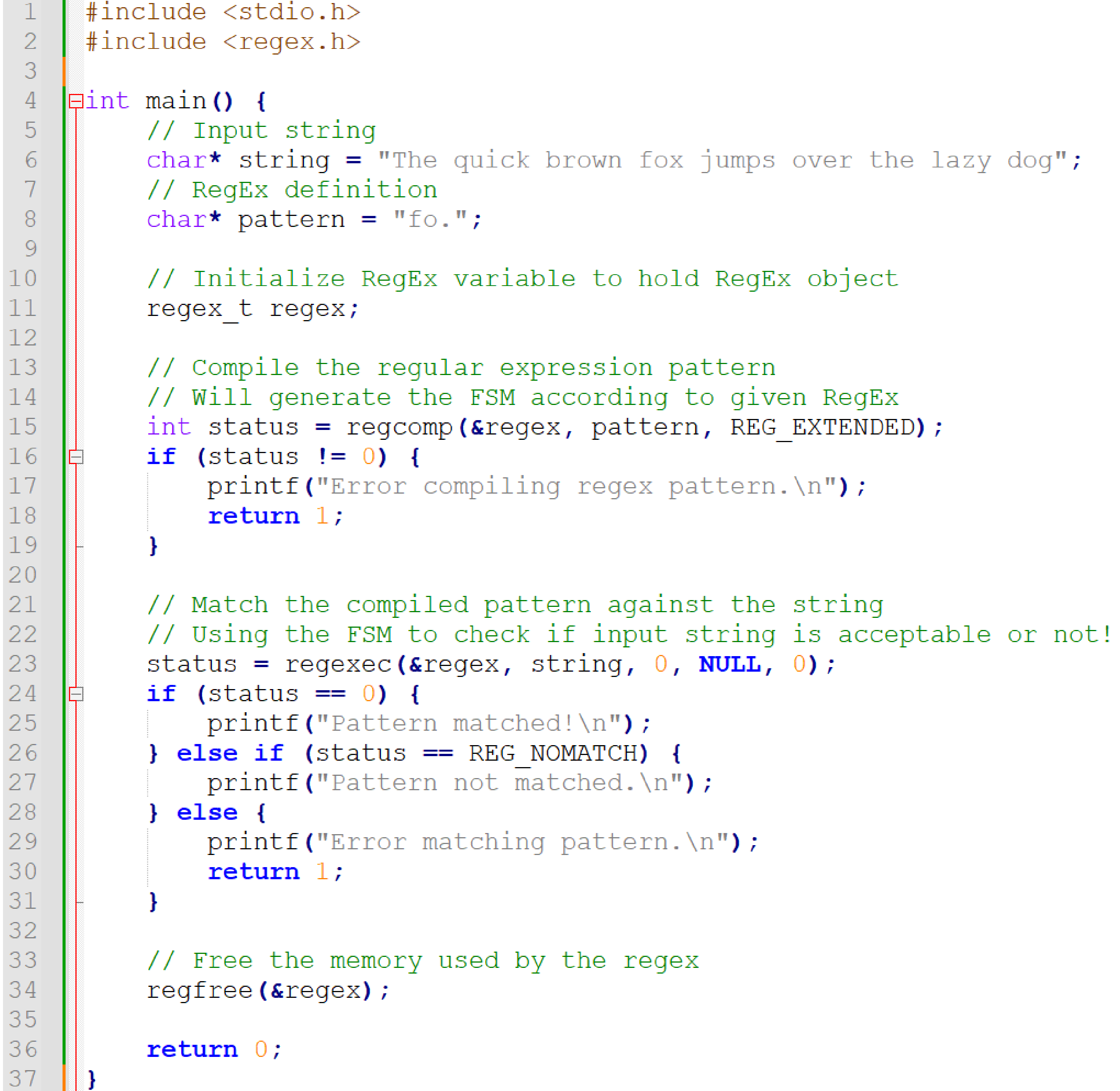
- Compiling a regular expression pattern into its equivalent FSM,
- Use this regular expression FSM on a given input string,
- Therefore, deciding if the input string is acceptable for this RegEx.
- Etc.

The RegEx library in C

The RegEx library is a library of functions and tools for working with regular expressions in C (among other languages) and provides a set of functions and tools for compiling, matching, and manipulating regular expressions

- Compiling a regular expression pattern into its equivalent FSM,
- Use this regular expression FSM on a given input string,
- Therefore, deciding if the input string is acceptable for this RegEx.
- Etc.

Will be useful to recognize keywords and other things during the Tokenization part of the compiler!



```c
#include <stdio.h>
#include <regex.h>

int main() {
    // Input string
    char* string = "The quick brown fox jumps over the lazy dog";
    // RegEx definition
    char* pattern = "fo.";

    // Initialize RegEx variable to hold RegEx object
    regex_t regex;

    // Compile the regular expression pattern
    // Will generate the FSM according to given RegEx
    int status = regcomp(&regex, pattern, REG_EXTENDED);
    if (status != 0) {
        printf("Error compiling regex pattern.\n");
        return 1;
    }

    // Match the compiled pattern against the string
    // Using the FSM to check if input string is acceptable or not!
    status = regexec(&regex, string, 0, NULL, 0);
    if (status == 0) {
        printf("Pattern matched!\n");
    } else if (status == REG_NOMATCH) {
        printf("Pattern not matched.\n");
    } else {
        printf("Error matching pattern.\n");
        return 1;
    }

    // Free the memory used by the regex
    regfree(&regex);

    return 0;
}
```

More stuff about RegEx

For curiosity, but probably very niche in practice.

The curly braces in RegEx

In RegEx, the **curly braces ({})** are used to **specify how many times a preceding element should be matched**.

- **{n}** matches the preceding element exactly n times,
- **{n,m}** matches the preceding element between n and m times (inclusive),
- and **{n,}** matches the preceding element at least n times.

For instance,

- a{3} matches the string "aaa", but not "aa" or "aaaa".
- a{2,4} matches the string "aa", "aaa", or "aaaa", but not "a" or "aaaaa".
- \d{2,} matches any string that contains at least two digits, such as "123", "45", or "6789".

The boundary \b

Definition (The boundary \b):

The metacharacter \b matches a word boundary.

It is a position **between a word character and a non-word character**, or **between a non-word character and a word character**.

For example, the regex pattern \bcat\b matches the word "cat" when it occurs on its own as a separate word, but not when it appears as part of another word like "category" or "scat":

- “The cat is black and white” works, so does “I love my cat”.
- “The category of my work is computer science” does not.

The \b can then be used to recognize while in “while(something…)”.

Some additional practice

From past year homeworks

You are working for a survey company in Singapore. You will be sending forms to a vast quantity of SG-based users, asking for their names and phone numbers, along with many survey questions regarding various topics. Every survey form will require the participants to enter their phone number, as a string of digits.

Some algorithm will be used to remove all the whitespaces in the string entered by the user for you, and you should expect one of the three string formats below after the string has been cleaned. After cleaning the strings will look like “63036600”, “+6563036600”, or “006563036600”. Possibly with other digits than those shown.

Question: Can you write a RegEx for checking if the phone number consists of eight digit and maybe a country code +65 or 0065 that might have been added to the phone number?

- We expect the users to enter only digits, no phone numbers with more than 8 digits (unless country codes are used), and no country codes other than +65 or 0065.
- A phone number not using 8 digits (before a country code is added) is invalid.
- Any other country code than +65 or 0065 should be rejected.
- We shall only check for valid characters: it is ok if our RegEx does not catch a phone number +6500000000, which is obviously fake.

Let us consider strings consisting of 0s and 1s only.

We would like to check for strings that have the same digit in their first, third, fifth, etc. location. Have a look at the table below for some examples of acceptable and not acceptable strings.

Question: Your friend Chris claims that it is impossible to describe this pattern using a RegEx. Do you agree with him and can explain why this is not possible? Or can you provide a RegEx that works for this task and prove Chris wrong?

String	Acceptable?
1	Yes
101	Yes
01000101	Yes
100110	No
101011100	No