Restricted

50.051 Programming Language Concepts

W9-S2 Regular Expressions and Pattern Recognition in C

Matthieu De Mari

Restricted

Restricted

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 us to recognize substrings of text that exhibit a certain pattern (they consist of a certain keyword, start with “, etc.).

Restricted

Restricted

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.

Restricted

Restricted

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

Can we use FSM to recognize to implement RegEx?

Yes!

Our Finite State Machines (FSM) can 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.**

Restricted

Restricted

Technically, we have already played with RegEx FSMs! (Remember this activity).

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, if and only if the string *x* contains the substring ZAM.

Restricted

Restricted

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

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

Restricted

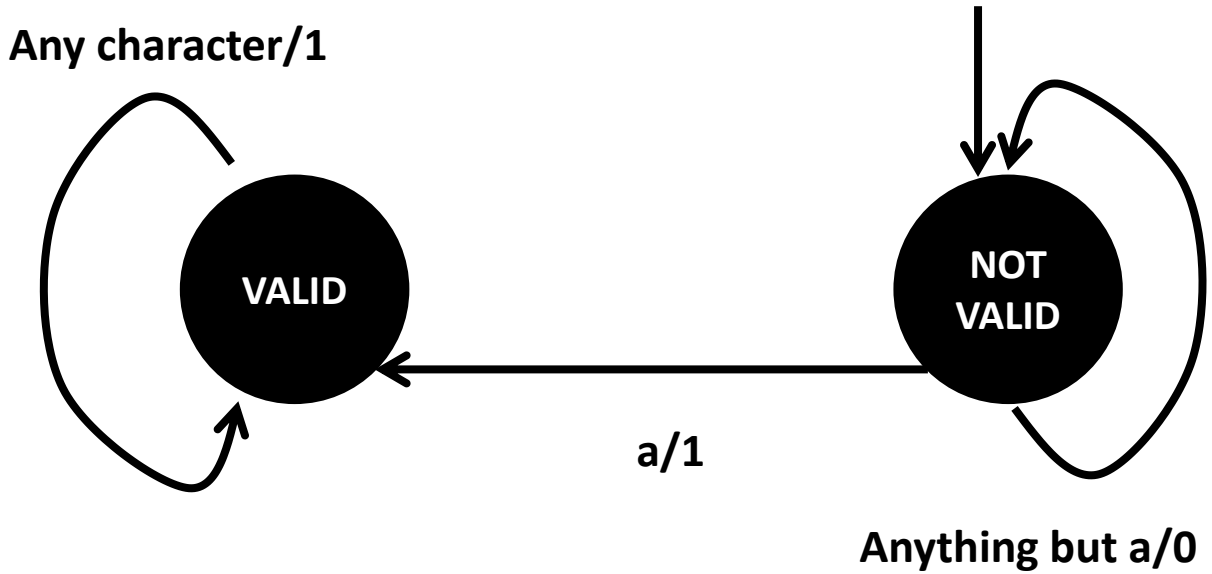
Restricted

The most basic RegEx

This can be simply implemented with the FSM shown.

The FSM uses outputs and no stopping states, 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.



Restricted

```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;
}
```

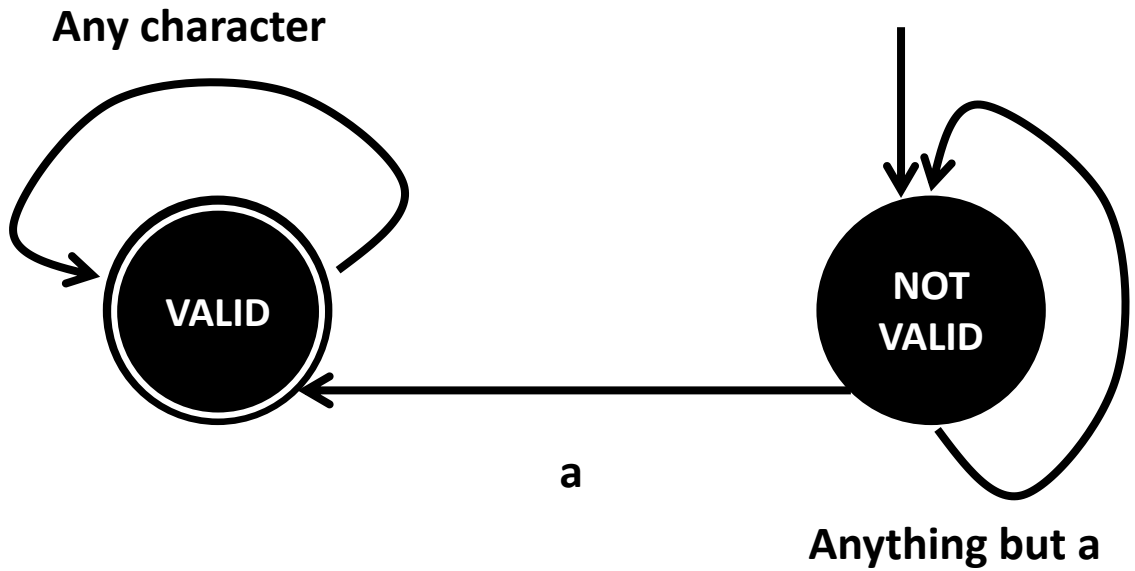
Restricted

The most basic RegEx

This can be also be implemented as an FSM with 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.



Restricted

```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;
}
```

Restricted

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 checking the next input character".

Restricted

Restricted

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 a path in the FSM that satisfies certain conditions in terms of stopping states or valid outputs being produced.

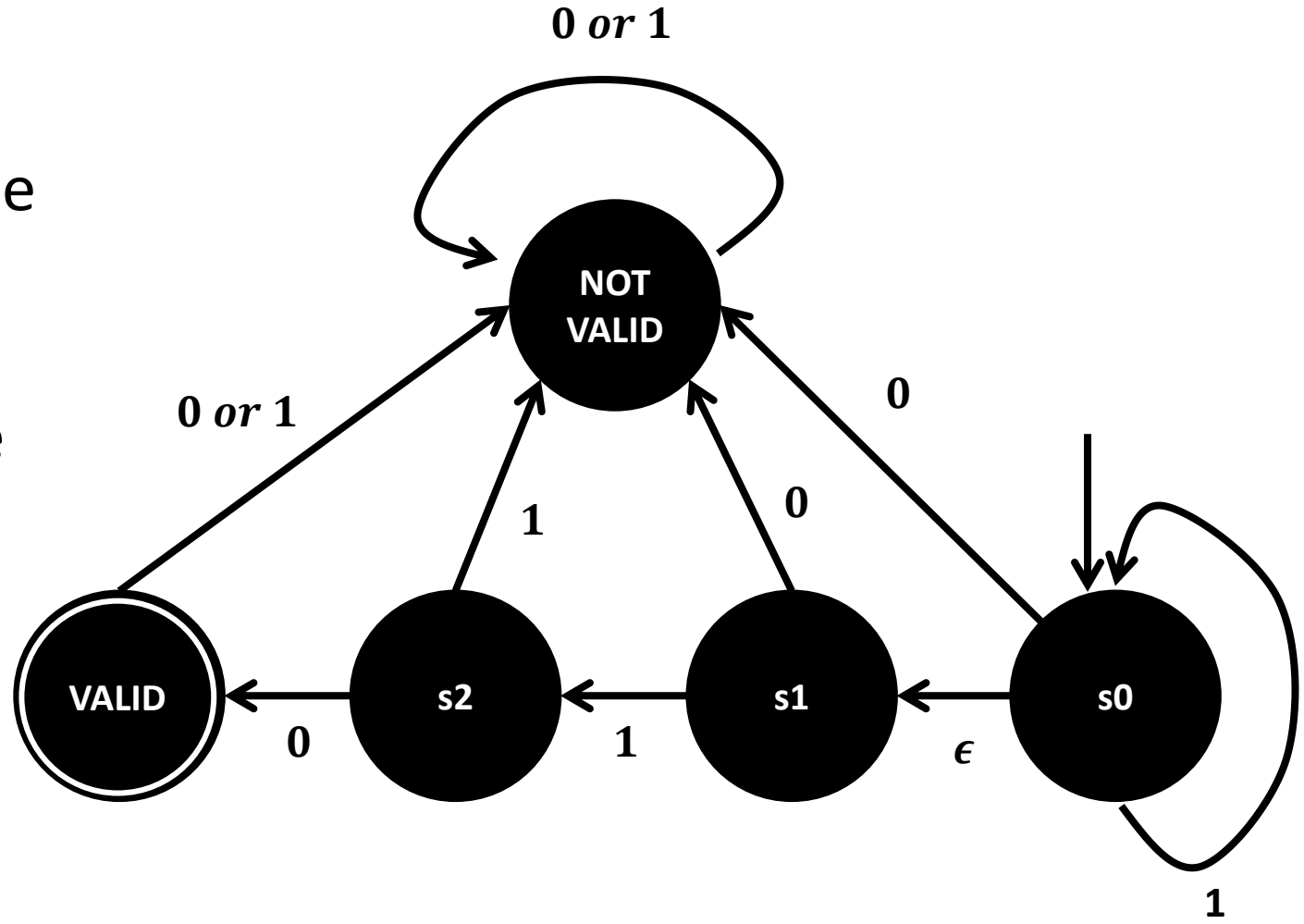
Restricted

Restricted

An example of a use for Epsilon

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

Question: What are acceptable inputs for this NDFSM?



Restricted

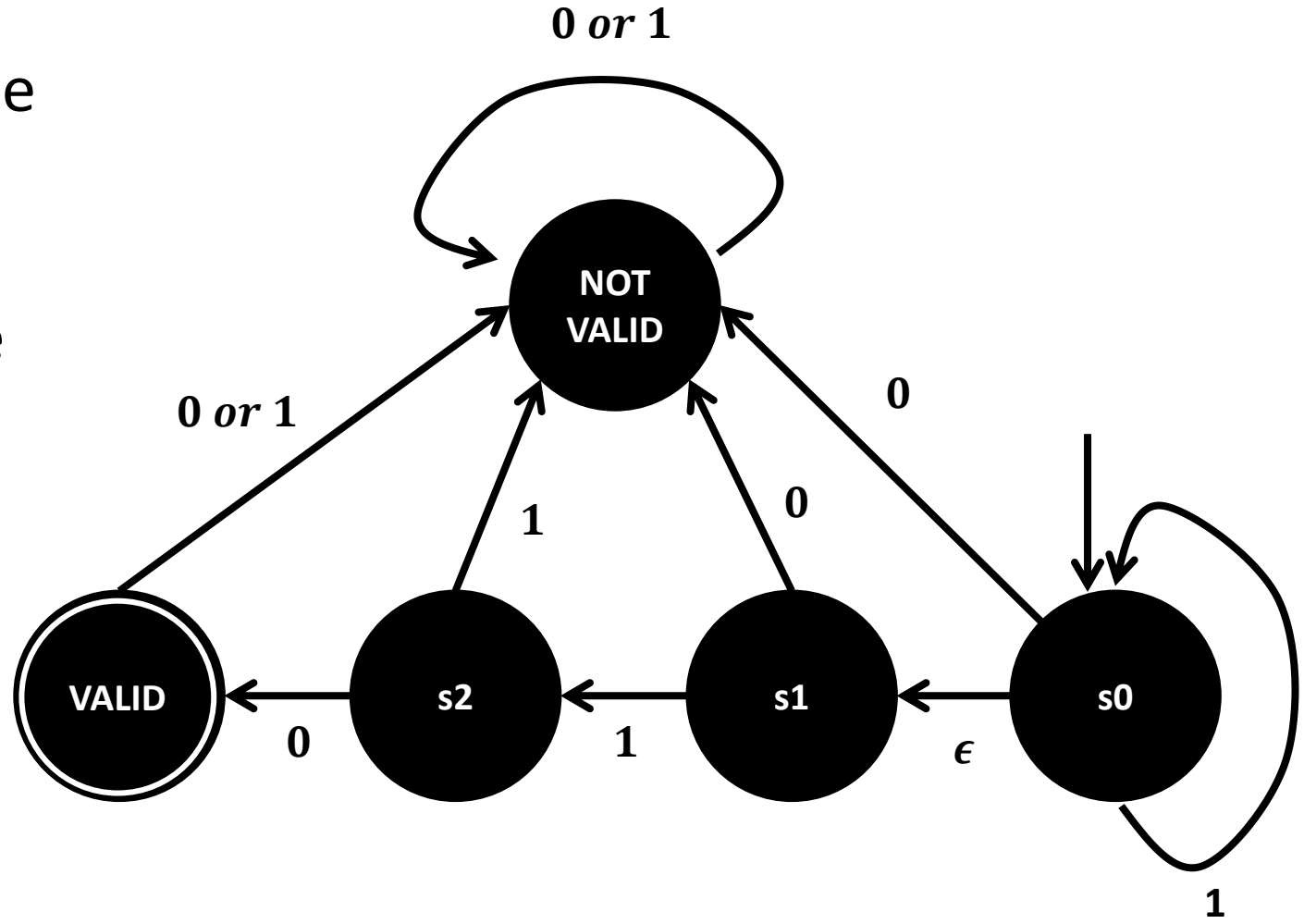
Restricted

An example of a use for Epsilon

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.)



Restricted

Restricted

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 is 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).

Restricted

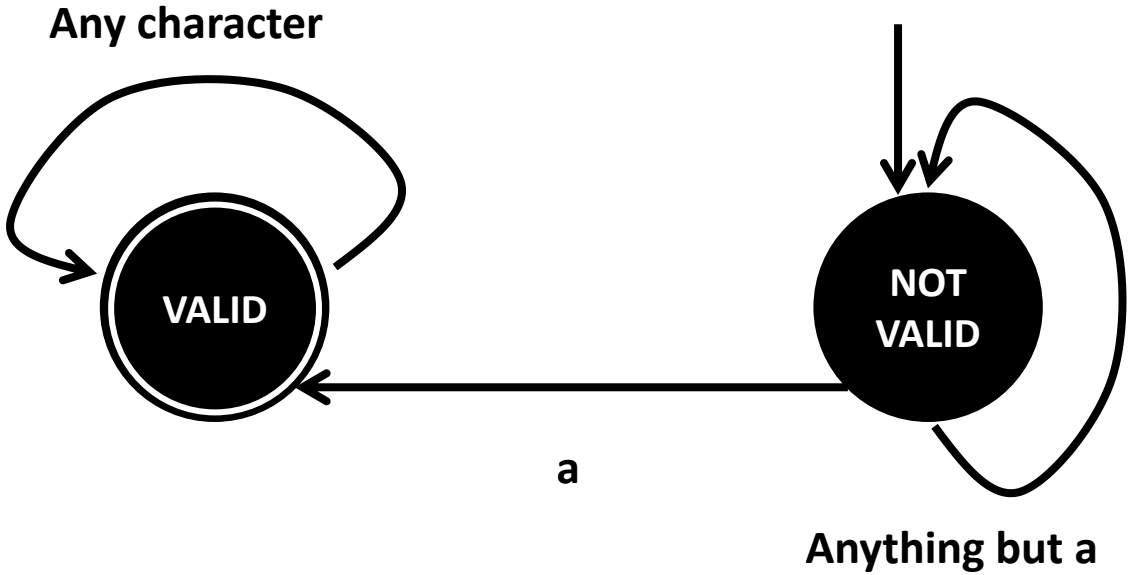
Restricted

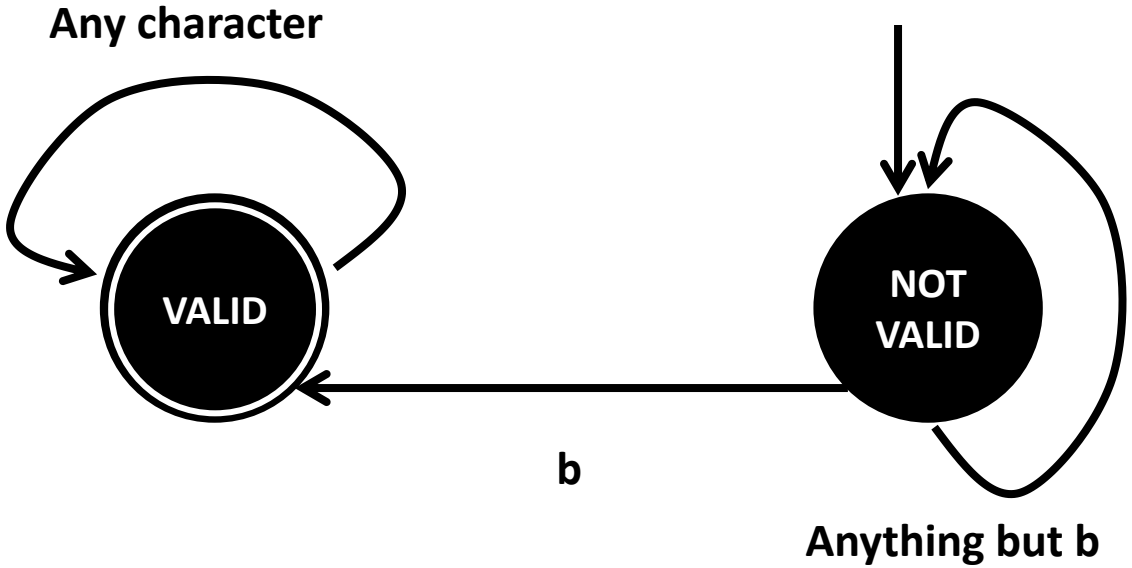
Concatenation of RegEx

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.





Restricted

Restricted

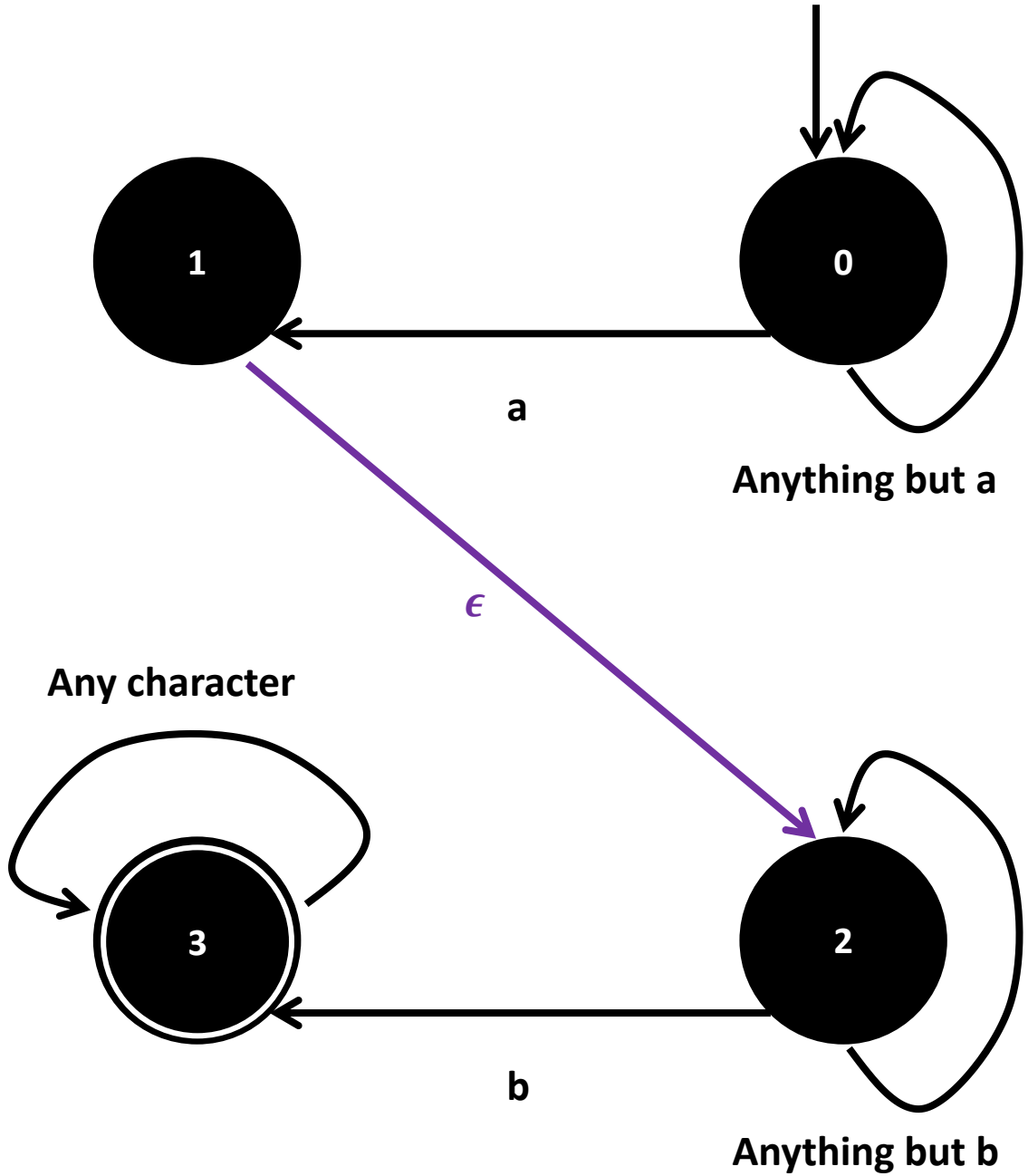
Concatenation of RegEx

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”**, can then be built **by combining the two FSMs, connecting them with an Epsilon transition.**



Restricted

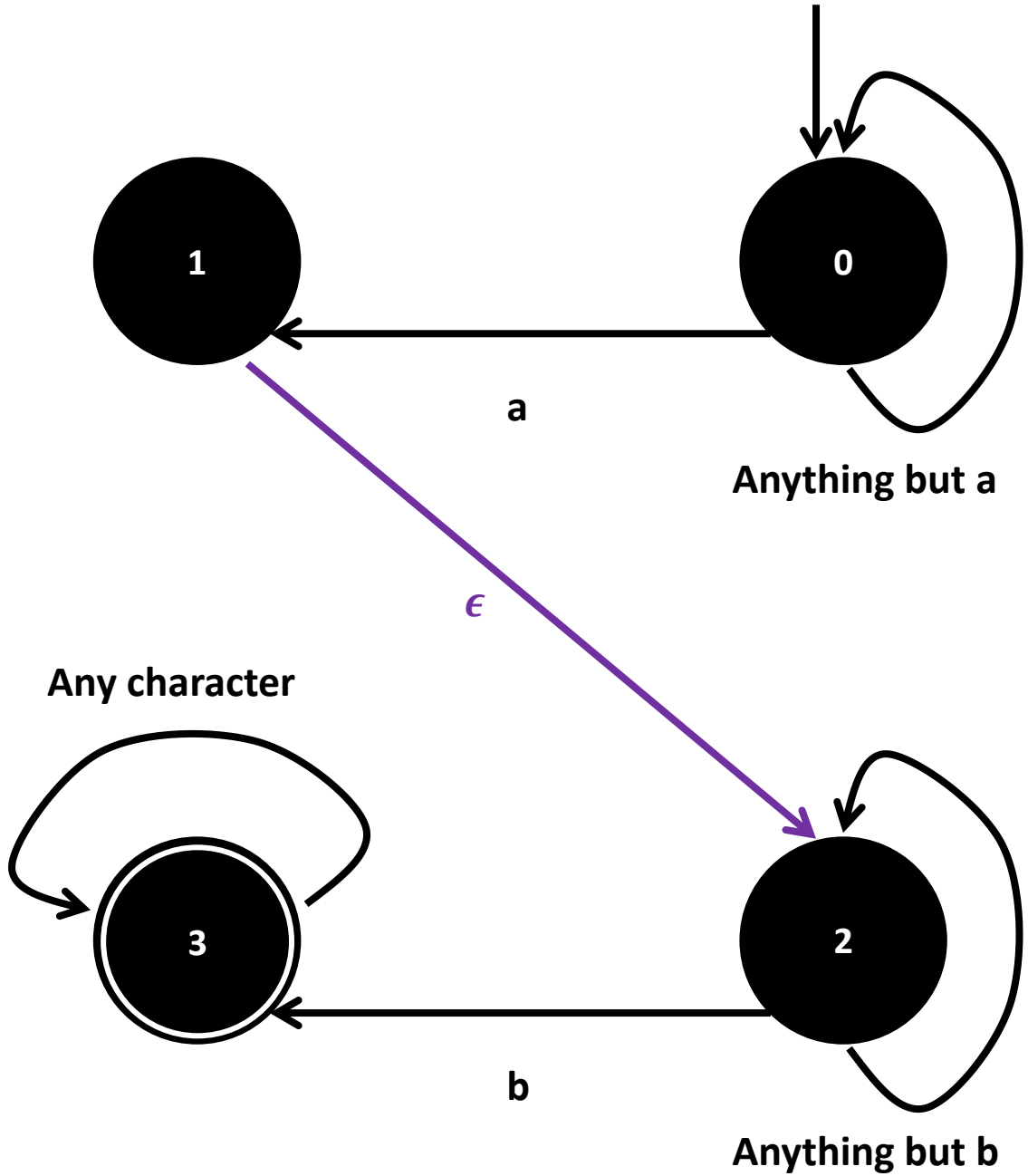
Restricted

Concatenation of RegEx

Quick note: It is true that we could have technically

- merged both states 1 and 2 together,
- and gotten rid of the Epsilon transition here,

but that might not always be the case.



Restricted

Restricted

What we have done in W8S3 Practice 2!

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, if and only if the string *x* contains the substring ZAM.

This is something we could do here!

Restricted

Restricted

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”** can be used to define two regular language consisting of the two strings “a” and “b”.

It then checks for strings that contains either a or b, or both.

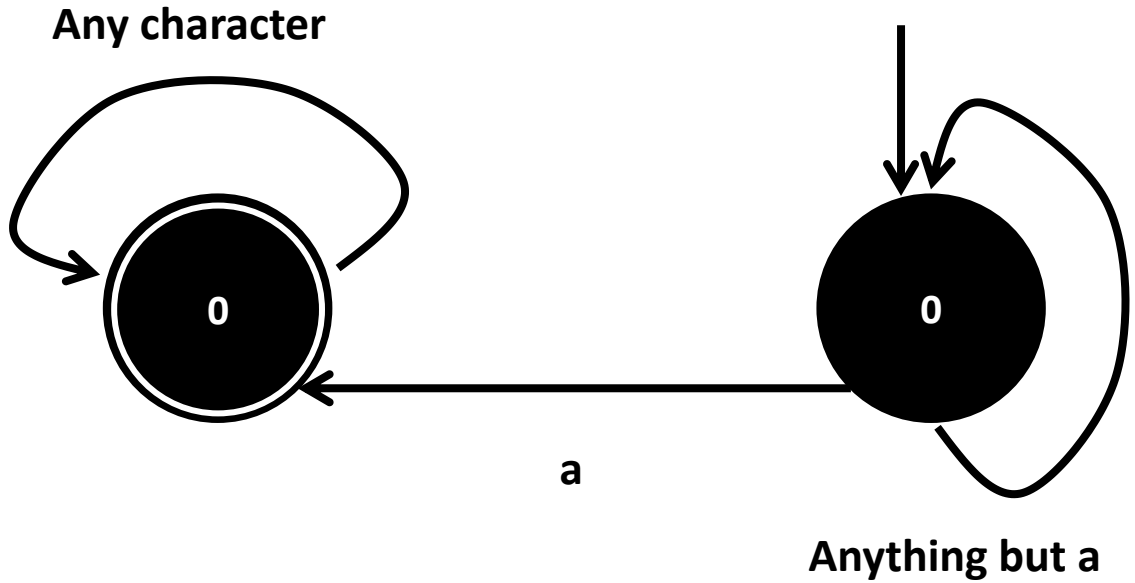
Restricted

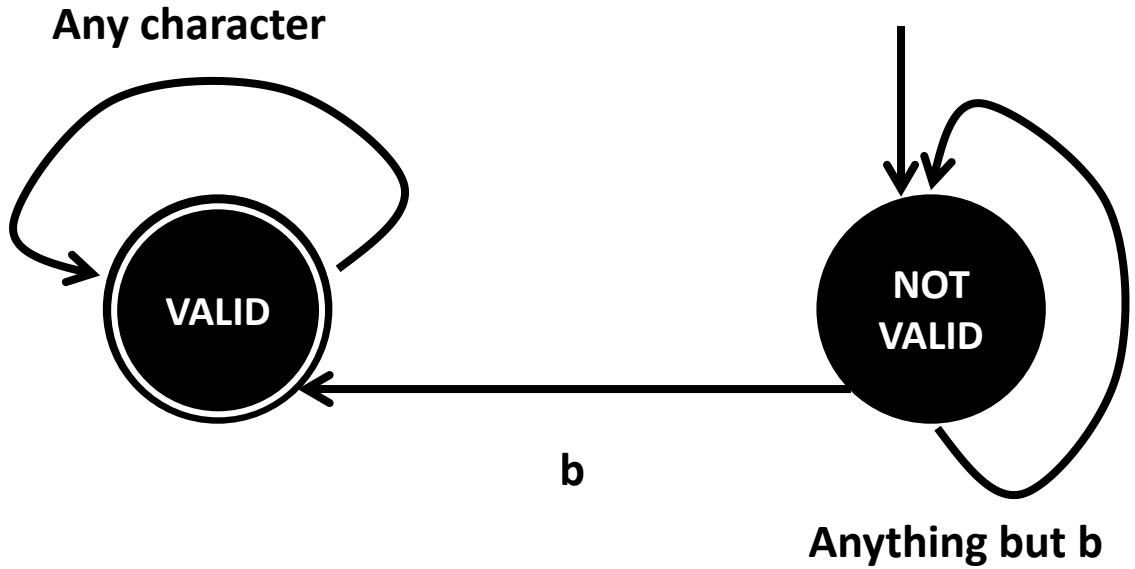
Restricted

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.





Restricted

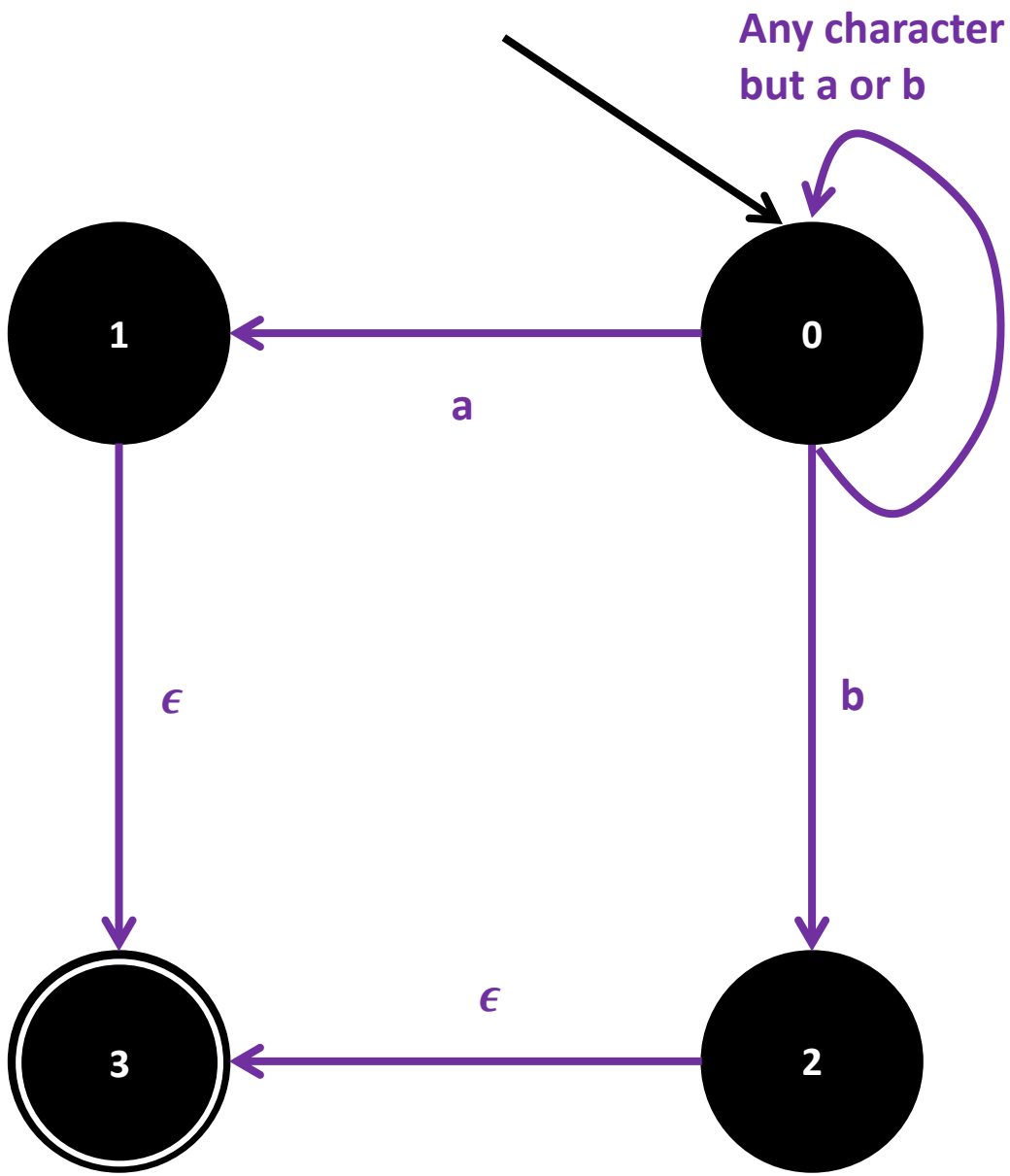
Restricted

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.

(Removing some self-links on states 1, 2, 3, for simplicity.)



Restricted

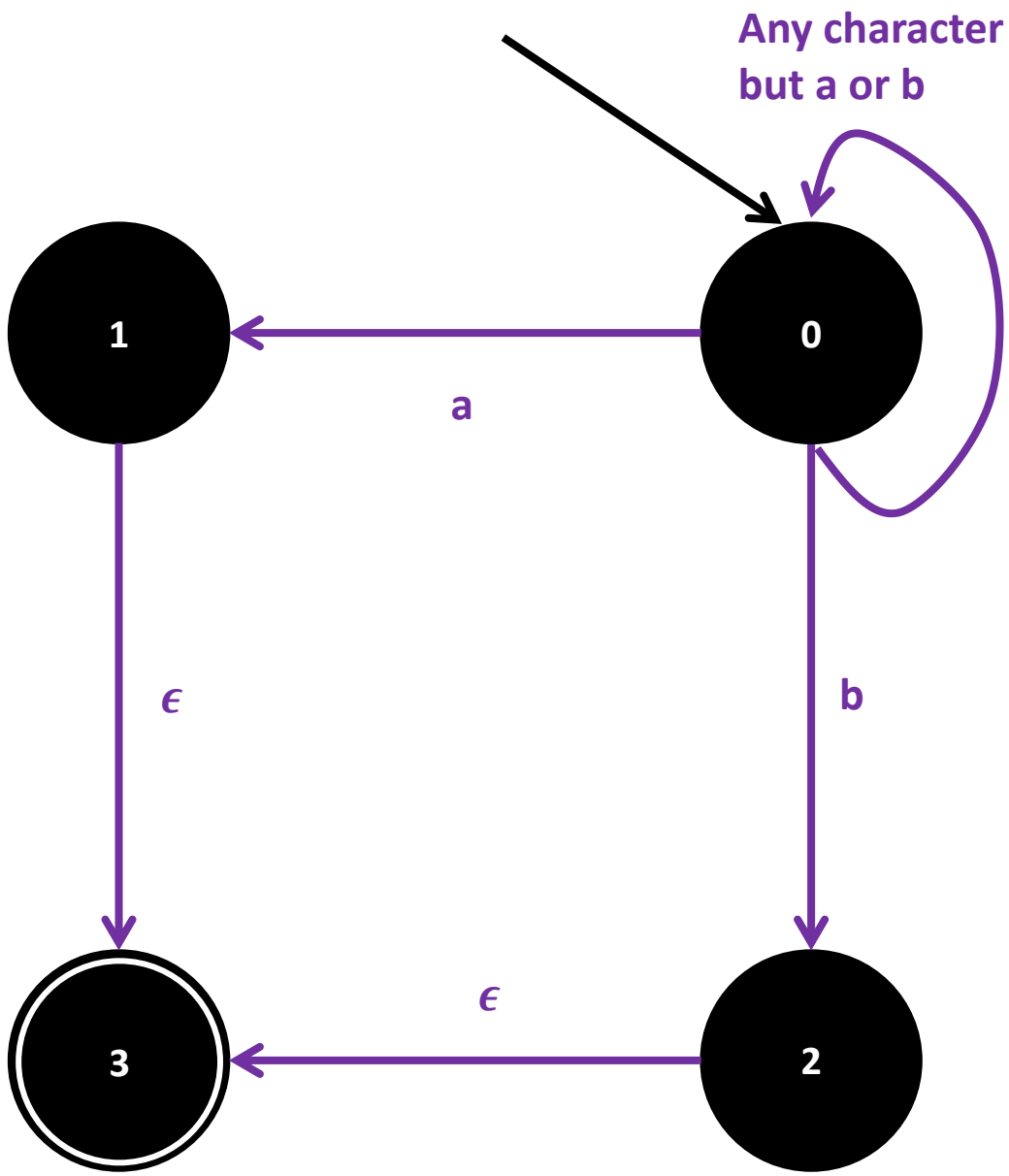
Restricted

Choice of RegEx

Quick note: Again, it is true that we could technically have

- merged both states 1, 2 and 3 together,
- and gotten rid of the Epsilon transition here,

but that might not always be the case.



Restricted

Restricted

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).**

Restricted

Restricted

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 **"(ac)".**

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

Restricted

Restricted

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**, i.e. “”, “a”, “aa”, “aaa”, “aaaaaa”, etc.

It is important to note that **“a*” does not match any other characters**. This means that “aaaab” will not be accepted by “a*”.

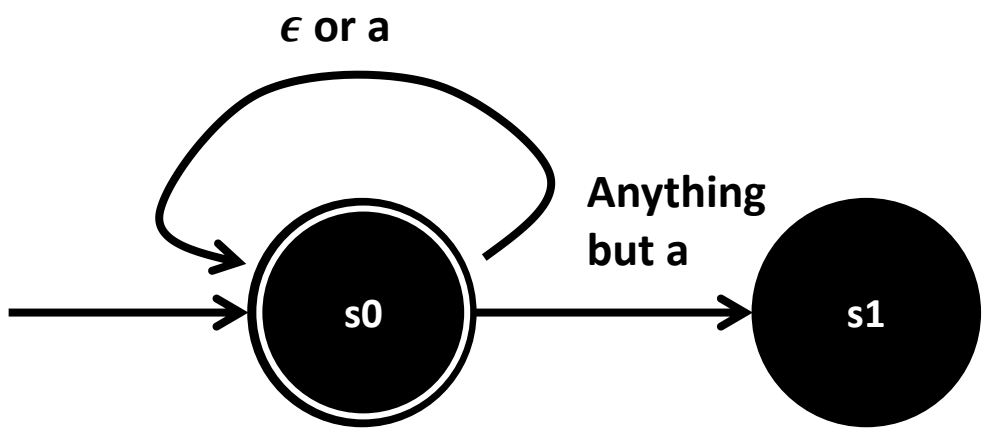
Restricted

Restricted

Closure or Kleene operator

The FSM for “a*” 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*”.



Restricted

Restricted

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 consists of **all the strings containing one or more repetitions of the character a**, i.e. “a”, “aa”, “aaa”, “aaaaa”, etc.

Again, it is important to note that **“a+” does not match any other characters**. This means that “aaaab” will not be accepted by “a+”.

Restricted

Restricted

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 operator) ?** is used to define the RegEx **“a?”**.

Its regular language consists of **all the strings containing zero or one repetitions of the character a**, i.e. “” and “a” only.

It is important to note that **“a?” does not match any other characters**. This means that “ab” will not be accepted by “a+”.

Restricted

Restricted

Quiz Time!

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

A. (c|b)at
B. (c|b)?at
C. (c+b)at
D. (c|b)+(at)

Restricted

Restricted

Quiz Time!

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

A. (c|b)at

B. (c|b)?at

C. (c+b)at

D. (c|b)+at

Restricted

Restricted

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

Restricted

Restricted

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

Restricted

Restricted

Quiz Time!

Which regular expression would match the strings "ab", "abb", "abbb", and so on, but no other strings?

A. abb*

B. ab

C. ab+

D. a+ab

Restricted

Restricted

Quiz Time!

Which regular expression would match the strings "ab", "abb", "abbb", and so on, but no other strings?

A. abb* (but is considered malpractice)

B. ab

C. ab+

D. a+ab

Restricted

Restricted

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

Restricted

Restricted

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\

Restricted

Restricted

The Kleene Theorem

Theorem (The Kleene Theorem):

The three fundamental operators we have defined earlier:

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

can be used to define every other operation in RegEx.

Operator precedence is RegEx is then defined as:

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

For instance,

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

Restricted

Restricted

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).

Restricted

Restricted

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_].

Restricted

Restricted

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.

Restricted

Restricted

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 (^) matches the beginning of a string.
- **$**: The dollar sign ($) matches 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$ matches the string "abc", but not "aabc", "abcc", or "dabc".
- The regex ^abc matches "abc" at the beginning of a string, but not "aabc" or "abcc".
- The regex abc$ matches "abc" at the end of a string, but not "abcc" or "dabc".

Restricted

Restricted

Negating a grouping

A quick note on the Caret symbol ^:

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

Restricted

Restricted

Quiz time!

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

A. “”
B. “B”
C. “ABC”
D. “A1B2C3”
E. “a”
F. “abC”
G. “123”

Restricted

Restricted

Quiz time!

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

A. “”

B. “B”

C. “ABC”

D. “A1B2C3”

E. “a”

F. “abC”

G. “123”

Restricted

Restricted

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

Restricted

Restricted

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

Restricted

Restricted

Quiz time!

What does the regular expression \s*cat*\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

Restricted

Restricted

Quiz time!

What does the regular expression \s*cat*\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

Restricted

Restricted

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 contains one or more lowercase letters and optionally ends with a digit
D. Any string that contains exactly one lowercase letter followed by a digit

Restricted

Restricted

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 contains one or more lowercase letters and optionally ends with a digit**
D. Any string that contains exactly one lowercase letter followed by a digit

Restricted

Restricted

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".

Restricted

Restricted

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,
- “The category of my work is irrelevant” does not.

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

Restricted

Restricted

The beginning of a string \A

Definition (The beginning of a string \A):

The metacharacter **\A** matches the start of a string.

It signifies the beginning of the string and is similar to the caret (^).

However, **unlike the caret, which would match the beginning of a line in in a string with multiple lines, \A matches only at the very beginning of the string.**

For example, the regex pattern \Ahello matches the word "hello" only if it appears at the very beginning of the string, and not anywhere else in the string.

Restricted

Restricted

The end of a string \Z

Definition (The end of a string \Z):

The metacharacter **\Z** matches the end of a string.

It signifies the end of the string and is similar to the dollar sign ($)

However, **unlike the dollar sign, which matches the end of a line in in a string with multiple lines, \Z matches only at the very end of the string.**

For example, the regex pattern world\Z matches the word "world" only if it appears at the very end of the string, and not anywhere else in the string.

If the string ends with a newline character, \Z matches just before that newline character.

Restricted

Restricted

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,
- Matching a regular expression FSM against a given string,
- Searching a string for the first occurrence of a regular expression pattern,
- Splitting a string and extracting substrings based on a regular expression pattern,
- Etc.

Restricted

Restricted

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,
- Matching a regular expression FSM against a given string,
- Searching a string for the first occurrence of a regular expression pattern,
- Splitting a string and extracting substrings based on a regular expression pattern,
- Etc.

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

Restricted

```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;
}
```