Computación y Estructuras Discretas III

Andrés A. Aristizábal P.
aaaristizabal@icesi.edu.co

Departamento de Computación y Sistemas Inteligentes

2024-2

Agenda del día

1. **Operations Using Languages**
   - Concatenation
   - Power
   - Kleene Closure
   - Reverse
2. **Regular languages and expressions**
   - Regular Languages
   - Regular Expressions
3. **Regular expressions in Python**
   - Exercises in Python

Operations as Sets

Definition

Since languages over Σ are subsets of Σ^*, the usual operations between sets are also valid operations between languages. Then, if A and B are languages over Σ ($A, B \subseteq \Sigma^*$), then the following are also languages over Σ :

$A \cup B$	*Union*
$A \cap B$	*Intersection*
$A - B$	*Difference*
$\overline{A} = \Sigma^* - A$	*Complement*

These operations between languages are called boolean operations in order to differentiate them from the linguistic operations, which are extensions to the languages of the operations between strings.

Agenda

Concatenation

Definition

The concatenation of two languages A and B over Σ, represented as $A \cdot B$ or simply AB is defined as:

Concatenation

Definition

The concatenation of two languages A and B over Σ, represented as $A \cdot B$ or simply AB is defined as:

$$AB = \{uv \mid u \in A, v \in B\}$$

Concatenation

Definition

The concatenation of two languages A and B over Σ, represented as $A \cdot B$ or simply AB is defined as:

$$AB = \{uv \mid u \in A, v \in B\}$$

In general, $AB \neq BA$

Properties over Concatenation

Definition

Given languages A, B, C *over* Σ, *i.e.* $A, B, C \subseteq \Sigma^*$. *Then*

Properties over Concatenation

Definition

Given languages A, B, C over Σ, i.e. $A, B, C \subseteq \Sigma^*$. Then

1. $A \cdot \emptyset = \emptyset \cdot A = \emptyset$.
2. $A \cdot \{\lambda\} = \{\lambda\} \cdot A = A$.
3. *Associative property,*
$$A \cdot (B \cdot C) = (A \cdot B) \cdot C.$$
4. *Distributivity of concatenation with respect to union,*
$$A \cdot (B \cup C) = A \cdot B \cup A \cdot C.$$
$$(B \cup C) \cdot A = B \cdot A \cup C \cdot A.$$
5. *Generalized distributivity property. If $\{B_i\}_{i \in I}$ is any family of languages over Σ, then*
$$A \cdot \bigcup_{i \in I} B_i = \bigcup_{i \in I} (A \cdot B_i),$$
$$\left(\bigcup_{i \in I} B_i \right) \cdot A = \bigcup_{i \in I} (B_i \cdot A).$$

Agenda

1. **Operations Using Languages**
   - Concatenation
   - Power
   - Kleene Closure
   - Reverse
2. **Regular languages and expressions**
   - Regular Languages
   - Regular Expressions
3. **Regular expressions in Python**
   - Exercises in Python

Power

What is the power of a language?

Definition

Given a language A over Σ, ($A \subseteq \Sigma^*$), and a natural number $n \in \mathbb{N}$, A^n is defined in the following way:

$$\begin{aligned} A^0 &= \{\lambda\}, \\ A^n &= \underbrace{AA\cdots A}_{n \text{ times}} = \{u_1 \cdots u_n \mid u_i \in A, \text{ for all } i,\ 1 \leq i \leq n\}. \end{aligned}$$

A^2 is the set of double string concatenations of A, A^3 the set of triple string concatenations of A and in general A^n is the set of n string concatenations of A.

Agenda

Kleene Closure

Definition

The Kleene closure of A, $A \subseteq \Sigma^*$, is the union of all powers of A and it is represented by A^*.

$$\begin{aligned} A^* &= \text{set of all string concatenations of } A, \\ &\quad \text{including } \lambda \\ &= \{u_1, \cdots u_n \mid u_i \in A,\ n \geq 0\} \end{aligned}$$

Positive Closure

Definition

The positive closure of a language A, $A \subseteq \Sigma^*$, is the union of all powers of A, without including λ and is represented by A^+.

$$\begin{aligned} A^+ &= \text{set of all string concatenations of } A, \\ &\quad \text{without including } \lambda \\ &= \{u_1, \cdots u_n \mid u_i \in A,\ n \geq 1\} \end{aligned}$$

Properties

Which are the properties of $*$ and $+$?

Definition

1. $A^+ = A^* \cdot A = A \cdot A^*$.
2. $A^* \cdot A^* = A^*$.
3. $(A^*)^n = A^*$, *for all* n, $n \geq 1$.
4. $(A^*)^* = A^*$.
5. $A^+ \cdot A^+ \subseteq A^+$.
6. $(A^*)^+ = A^*$.
7. $(A^+)^* = A^*$.
8. $(A^+)^+ = A^+$.
9. *If* A *and* B *are languages over* Σ^*, *then* $(A \cup B)^* = (A^* B^*)^*$.

Agenda

Reverse

Definition

Given A, a language over Σ, we define A^R in the following way:

$$A^R = \{u^R \mid u \in A\}$$

A^R is the inverse of A.

Agenda

1. Operations Using Languages
   - Concatenation
   - Power
   - Kleene Closure
   - Reverse
2. **Regular languages and expressions**
   - Regular Languages
   - Regular Expressions
3. Regular expressions in Python
   - Exercises in Python

Regular Languages

Regular languages from a given alphabet Σ are all those languages that can be built from the basic languages $\emptyset, \{\lambda\}, \{a\}, a \in \Sigma$, using union, concatenation and Kleene closure.

Regular Languages

Regular languages from a given alphabet Σ are all those languages that can be built from the basic languages $\emptyset, \{\lambda\}, \{a\}, a \in \Sigma$, using union, concatenation and Kleene closure.

Given an alphabet Σ

Definition

1. $\emptyset, \{\lambda\}, \{a\}$, *for all* $a \in \Sigma$, *are regular languages over* Σ. *These are the so called basic regular languages.*
2. *If* A *and* B *are regular languages over* Σ, *then the following are also regular:*

$$\begin{array}{ll} A \cup B & \textit{(union)}, \\ A \cdot B & \textit{(concatenation)}, \\ A^* & \textit{(Kleene closure)}. \end{array}$$

Regular Languages

Σ and Σ^* are both regular languages over Σ. Union, concatenation and Kleene closure are known as regular operations.

Example

Given $\Sigma = \{a, b\}$. The following ones are regular languages over Σ :

1. *The language A of all strings that have just one a :*
$$A = \{b\}^* \cdot \{a\} \cdot \{b\}^*.$$
2. *The language B of all strings that start with b :*
$$B = \{b\} \cdot \{a, b\}^*.$$
3. *The language C of all strings that contain the string ba :*
$$C = \{a, b\}^* \cdot \{ba\} \cdot \{a, b\}^*.$$
4. $(\{a\} \cup \{b\}^*) \cdot \{a\}$.
5. $[(\{a\}^* \cup \{b\}^*) \cdot \{b\}]^*$.

Agenda

1. Operations Using Languages
   - Concatenation
   - Power
   - Kleene Closure
   - Reverse
2. **Regular languages and expressions**
   - Regular Languages
   - Regular Expressions
3. Regular expressions in Python
   - Exercises in Python

Regular Expressions

Regular expression is a sequence of symbols and characters expressing a **Regular Language**, they are used as strings or patterns to be searched for within a longer piece of text.

Regular Expressions

Definition

1. *Basic regular expressions:*

   $\emptyset$ *is a regular expression that represents the language* $\emptyset$
   λ *is a regular expression that represents the language* $\{\lambda\}$.
   a *is a regular expression that represents the language* $\{a\}$, $a \in \Sigma$.

2. *If* R *and* S *are regular expressions over* Σ, *the following ones are also regular:*

$$(R)(S)$$
$$(R \cup S)$$
$$(R)^*$$

Example

Example

Given the alphabet $\Sigma = \{a, b, c\}$,

$$(a \cup b^*)a^*(bc)^*$$

is the regular expression that represents

$$(\{a\} \cup \{b\}^*) \cdot \{a\}^* \cdot \{bc\}^*$$

An overview of regular expressions

In programming, a regular expression is

- A pattern of text that consists of ordinary characters and special characters known as metacharacters.
- e.g. `a\w*`

What are literals?

- Are the simplest form of pattern matching in regular expression.
- They will simply succeed whenever that literal is found.
- If we apply the regular expression fox to search the phrase The quick brown fox jumps over the lazy dog, we will find one match.
- We can also obtain several results instead of just one, if we apply the regular expression `be` to the following phrase To be, or not to be.

What are metacharacters?

- A metacharacter is a character that has a special meaning during pattern processing.
- We use metacharacters in regular expressions to define the search criteria and any text manipulations.
- The simplest example of a metacharacter is the full stop.
- The full stop character matches any single character of any sort (apart from a newline).
- e.g. `.at` means: any letter, followed by the letter `a` followed by the letter `t`.

What are character classes?

- The character classes (also known as character sets) allow us to define a character that will match if any of the defined characters on the set is present.
- To define a character class, we should use the opening square bracket metacharacter [, then any accepted characters, and finally close with a closing square bracket].
- e.g. `licen[cs]e`
- It is possible to also use the range of a character. This is done by leveraging the hyphen symbol (-) between two related characters.
- e.g. To match any lowercase letter we can use `[a-z]`.
- e.g. To match any single digit we can define the character set `[0-9]`.

How to work with several character classes?

- The character classes' ranges can be combined to be able to match a character against many ranges.
- We just need to put one range after the other.
- e.g. If we want to match any lowercase or uppercase alphanumeric character, we can use `[0-9a-zA-Z]`
- This can be alternatively written using the union mechanism: `[0-9[a-z[A-Z]]]`.

What do we mean by negation of ranges?

- We can invert the meaning of a character set by placing a caret (^) symbol right after the opening square bracket metacharacter ([).
- If we have a character class such as `[0-9]` meaning any digit, the negated character class `[^0-9]` will match anything that is not a digit.

An overview of regular expressions

How to match against a set of regular expressions?

An overview of regular expressions

How to match against a set of regular expressions?

- This is accomplished by using the pipe symbol `|`.

An overview of regular expressions

How to match against a set of regular expressions?

- This is accomplished by using the pipe symbol `|`.
- It alternates between regular expressions.

An overview of regular expressions

How to match against a set of regular expressions?

- This is accomplished by using the pipe symbol `|`.
- It alternates between regular expressions.
- e.g. `yes|no`

How to match against a set of regular expressions?

- This is accomplished by using the pipe symbol `|`.
- It alternates between regular expressions.
- e.g. `yes|no`
- e.g. `yes|no|maybe`

An overview of regular expressions

What are quantifiers?

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions
- `*` 0 or more times

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions
- `*` 0 or more times
- `+` 1 or more times

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions
- `*` 0 or more times
- `+` 1 or more times
- `{n,m}` Between *n* and *m* times.

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions
- `*` 0 or more times
- `+` 1 or more times
- `{n,m}` Between *n* and *m* times.
  - `{n}` Exactly *n* times.

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions
- `*` 0 or more times
- `+` 1 or more times
- `{n,m}` Between *n* and *m* times.
  - `{n}` Exactly *n* times.
  - `{n,}` *n* or more times.

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions
- `*` 0 or more times
- `+` 1 or more times
- `{n,m}` Between *n* and *m* times.
  - `{n}` Exactly *n* times.
  - `{n,}` *n* or more times.
  - `{,n}` At most *n* times.

What are quantifiers?

- The mechanisms to define how a character, metacharacter, or character set can be repeated.

Which are the three basic quantifiers?

- `?` 0 or 1 repetitions
- `*` 0 or more times
- `+` 1 or more times
- `{n,m}` Between *n* and *m* times.
  - `{n}` Exactly *n* times.
  - `{n,}` *n* or more times.
  - `{,n}` At most *n* times.
- e.g. `cars?` Singular or plural.

Example

How to match a telephone number that can be in the format 555-555-555, 555 555 555, or 555555555.

What are greedy and reluctant quantifiers?

What are greedy and reluctant quantifiers?

- The greedy behavior of the quantifiers is applied by default in the quantifiers.

What are greedy and reluctant quantifiers?

- The greedy behavior of the quantifiers is applied by default in the quantifiers.
- A greedy quantifier will try to match as much as possible to have the biggest match result possible.

What are greedy and reluctant quantifiers?

- The greedy behavior of the quantifiers is applied by default in the quantifiers.
- A greedy quantifier will try to match as much as possible to have the biggest match result possible.
- The non-greedy behavior can be requested by adding an extra question mark to the quantifier.

What are greedy and reluctant quantifiers?

- The greedy behavior of the quantifiers is applied by default in the quantifiers.
- A greedy quantifier will try to match as much as possible to have the biggest match result possible.
- The non-greedy behavior can be requested by adding an extra question mark to the quantifier.
- For example, `??`, `*?` or `+?`.

What are greedy and reluctant quantifiers?

- The greedy behavior of the quantifiers is applied by default in the quantifiers.
- A greedy quantifier will try to match as much as possible to have the biggest match result possible.
- The non-greedy behavior can be requested by adding an extra question mark to the quantifier.
- For example, `??`, `*?` or `+?`.
- A quantifier marked as reluctant will behave like the exact opposite of the greedy ones.

What are greedy and reluctant quantifiers?

- The greedy behavior of the quantifiers is applied by default in the quantifiers.
- A greedy quantifier will try to match as much as possible to have the biggest match result possible.
- The non-greedy behavior can be requested by adding an extra question mark to the quantifier.
- For example, `??`, `*?` or `+?`.
- A quantifier marked as reluctant will behave like the exact opposite of the greedy ones.
- They will try to have the smallest match possible.

Example

Given the string `English → "Hello", → Spanish → "Hola".` *What would be the results of using the regular expressions* `".+"` *and* `".+?"` *over that string.*

What are boundary matchers?

What are boundary matchers?

- The boundary matchers are a number of identifiers that will correspond to a particular position inside of the input.

What are boundary matchers?

- The boundary matchers are a number of identifiers that will correspond to a particular position inside of the input.
- ^ Matches at the beginning of a line

What are boundary matchers?

- The boundary matchers are a number of identifiers that will correspond to a particular position inside of the input.
- `^` Matches at the beginning of a line
- `$` Matches at the end of a line

What are boundary matchers?

- The boundary matchers are a number of identifiers that will correspond to a particular position inside of the input.
- `^` Matches at the beginning of a line
- `$` Matches at the end of a line
- `\b` Matches a word boundary

What are boundary matchers?

- The boundary matchers are a number of identifiers that will correspond to a particular position inside of the input.
- `^` Matches at the beginning of a line
- `$` Matches at the end of a line
- `\b` Matches a word boundary
- `\B` The opposite of \b

What are boundary matchers?

- The boundary matchers are a number of identifiers that will correspond to a particular position inside of the input.
- `^` Matches at the beginning of a line
- `$` Matches at the end of a line
- `\b` Matches a word boundary
- `\B` The opposite of \b
- `\A` Matches the beginning of the input

What are boundary matchers?

- The boundary matchers are a number of identifiers that will correspond to a particular position inside of the input.
- `^` Matches at the beginning of a line
- `$` Matches at the end of a line
- `\b` Matches a word boundary
- `\B` The opposite of \b
- `\A` Matches the beginning of the input
- `\Z` Matches the end of the input

Example

Write a regular expression that will match lines that start with `Name:` *and make sure that after the name, there are only alphabetic characters or spaces until the end of the line.*

Example

What is the difference between the regular expressions `hello` *and* `\bhello\b`*?*

Regular expressions in Python

How are regular expressions supported in Python?

- They are supported by the `re` module.
- We only need to import it to start using it.

How to start using them to match a pattern?

- We compile a pattern with `pattern = re.compile(r'foo')`
- We can try to match it against a string `pattern.match("foo bar")`

What are the building blocks for Python Regex?

- RegexObject
  - Also known as Pattern Object.
  - Represents a compiled regular expression.
- MatchObject
  - Represents the matched pattern.

What is a RegexObject?

- Before matching patterns we need to compile the regex.
- The compilation produces a reusable pattern object.
- This object provides all the operations that can be done (i.e. matching a pattern and finding all substrings that match a particular regex).
- e.g. `pattern = re.compile(r'<HTML>')`
- `pattern.match("<HTML>")`

Regular expressions in Python

What are two ways of matching a pattern?

- We can compile a pattern, which gives us a RegexObject.
- We can just use the module operations.
- If we compile a pattern we are able to reuse it.
- e.g. `pattern = re.compile(r'<HTML>')`
- `pattern.match("<HTML>")`
- e.g. `re.match(r'<HTML>', "<HTML>")`

How to search for string that match a pattern?

- In python we have two operations match and search.

How does match work?

- This method tries to match the compiled pattern only at the beginning of the string.
- If there is a match, then it returns a MatchObject.
- It has two optional parameters, pos and endpos.
- pos determines the position from where to search the pattern in the string.
- endpos determines the position until where the pattern is searched in the string.
- `pattern.match(string,pos,endpos)`

Example

Given `pattern = re.compile(r'^<HTML>')` *what is the result of:*

Example

Given `pattern = re.compile(r'^<HTML>')` *what is the result of:*

- `pattern.match("<HTML>")`
- `pattern.match(" <HTML>")`
- `pattern.match(" <HTML>"[2:])`

Example

Given `pattern = re.compile(r'<HTML>$')` *what is the result of:*

- `pattern.match("<HTML> ",0,6)`
- `pattern.match("<HTML> "[:6])`

How does search work?

- This operation would be like the match of many languages.
- It tries to match the pattern at any location of the string and not just at the beginning.
- If there is a match, then it returns a MatchObject.
- The pos and endpos parameters have the same meaning as that in the match operation.
- pos determines the position from where to search the pattern in the string.
- endpos determines the position until where the pattern is searched in the string.
- `pattern.match(string,pos,endpos)`

Example

Given `pattern = re.compile(r"world")` *what is the result of:*

- `pattern.match("hello world")`
- `pattern.match("hola mundo ")`

Example

Given `pattern = re.compile(r'^<HTML>', re.MULTILINE)` *what is the result of:*

- `pattern.search("<HTML>")`
- `pattern.search(" <HTML>")`
- `pattern.search(" \n<HTML>")`
- `pattern.search(" \n<HTML>",3)`
- `pattern.search("</div></body>\n<HTML>",4)`
- `pattern.search(" \n<HTML>",4)`

How does findall work?

- It returns a list with all the non-overlapping occurrences of a pattern and not the MatchObject like search and match do.
- e.g. `pattern = re.compile(r"\w+")`
- `pattern.findall("hello world")`
- `['hello', 'world']`

Regular expressions in Python

Example

- `pattern = re.compile(r"(\w+) (\w+)")`
- `pattern.findall("Hello world hola mundo")`
- `[('Hello', 'world'), ('hola', 'mundo')]`

How does finditer work?

- Works essentially as as findall.
- It returns an iterator in which each element is a MatchObject.
- We can use the operations provided by this object.
- Useful when you need information for every match.

Regular expressions in Python

Example

- `pattern = re.compile(r"(\w+) (\w+)")`
- `it = pattern.finditer("Hello world hola mundo")`
- `match = it.next()`
- `match.groups()`
- `('Hello','world')`
- `match.span()`
- `(0,11)`

Which are some operations to modify strings?

Which are some operations to modify strings?

- split(string, maxsplit=0): A string can be split based on the matches of the pattern.
- sub(repl, string, count=0): Returns the resulting string after replacing the matched pattern in the original string with the replacement.

Example

- `pattern = re.compile(r"\W")`
- `pattern.split("Beautiful is better than ugly", 2)`
- `['Beautiful', 'is', 'better than ugly']`

Example

- `pattern = re.compile(r'[0-9]+)`
- `pattern.sub("-","order0, order1 order13")`
- `'order- order- order-'`

What is a MatchObject?

What is a MatchObject?

- An object that represents the matched pattern.
- We will get one every time you execute one of these operations:
  - match
  - search
  - finditer
- Provides a set of operations for working with the captured groups.

Which are the main operations for a MatchObject?

Which are the main operations for a MatchObject?

- group
- groups
- groupdict
- start
- end
- span

What is the group operation?

What is the group operation?

- Gives the subgroups of the match.
- If invoked with no arguments or zero, returns the entire match.
- If one or more group identifiers are passed, the corresponding groups' matches will be returned.

Regular expressions in Python

Example

- *pattern = re.compile("(\w+) (\w+)")*
- *match = pattern.search("Hello world")*
- *match.group() → 'Hello world'*
- *match.group(0) → 'Hello world'*
- *match.group(1) → 'Hello'*
- *match.group(2) → 'world'*
- *match.group(0,2) → ('Hello world','world')*

Example

- *Groups can be named.*
- *If the pattern has named groups, they can be accessed using the names or the index:*
- `pattern = re.compile(r"(?P<first>\w+) (?P<second>\w+)")`
- `match = pattern.search("Hello world")`
- `match.group('first')` → `'Hello'`
- `match.group(1)` → `'Hello'`
- `match.group(0,'first',2)` → `('Hello world','Hello','world')`

What is the groups operation?

What is the groups operation?

- It returns a tuple with all the subgroups in the match instead of giving one or some of the groups.

Regular expressions in Python

Example

- `pattern = re.compile("(\w+) (\w+)")`
- `match = pattern.search("Hello world")`
- `match.groups()` → `('Hello', 'world')`

What is the groupdict operation?

What is the groupdict operation?

- It is used in the cases where named groups have been used.
- It will return a dictionary with all the groups that were found.
- `pattern = re.compile(r"(?P<first>\w+) (?P<second>\w+)")`
- `match = pattern.search("Hello world")`
- `{'first': 'Hello', 'second': 'world'}`
- If there aren't named groups, then it returns an empty dictionary.

Regular expressions in Python

What about start, end and span operations?

What about start, end and span operations?

- start: returns the index where the pattern matched.
- end: returns the end of the substring matched by the group.
- span: gives a tuple with the values from start and end.

Agenda

1. Operations Using Languages
   - Concatenation
   - Power
   - Kleene Closure
   - Reverse
2. Regular languages and expressions
   - Regular Languages
   - Regular Expressions
3. **Regular expressions in Python**
   - Exercises in Python