CS112: Theory of Computation (LFA)

Lecture6: Regular Expressions

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

March 23, 2022

Table of contents

Section 1

Previously on CS112

Formal definition

Definition

We say that R is a regular expression if R is:

1. a for some a in the alphabet Σ
2. ϵ
3. $\emptyset$
4. $(R_1 \cup R_2)$ where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$ where R_1 and R_2 are regular expressions, or
6. (R_1^*) where R_1 is a regular expression

Examples

We assume $\Sigma = \{0, 1\}$:

1. $0^*10^* = \{w | \text{w contains a single 1}\}$
2. $\Sigma^*1\Sigma^* = \{w | \text{w hast at least one 1}\}$
3. $\Sigma^*001\Sigma^* = \{w | \text{w contains the string 001 as substring}\}$
4. $1^*(01^+)^* = \{w | \text{every 0 in w is followed by at least one 1}\}$
5. $(\Sigma\Sigma)^* = \{w | \text{w is a string of even length}\}$
6. $(\Sigma\Sigma\Sigma)^* = \{w | \text{w is a string with length multiple of 3}\}$
7. $01 \cup 10 = \{01, 10\}$
8. $(0 \cup \epsilon)1^* = 01^* \cup 1^*$
9. $1^*\emptyset = \emptyset$

Section 2

Context setup

Context setup

Corresponding to Sipser 1.3

Context setup

We continue with equivalence proof and then relax with some practical RegEx

Section 3

Equivalence with finite automata

Few remarks

- Regular expressions and finite automata are equivalent in their descriptive power even if they are different
- Any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa
- Recall that a regular language is one that is recognized by some finite automaton

Equivalence with finite automata

Theorem

A language is regular if and only if some regular expression describes it.

Proof.

Two directions proof. We state and prove each direction as a separate lemma □

Equivalence with finite automata

Lemma

If a language is described by a regular expression, then it is regular ($\Leftarrow$).

Lemma

If a language is regular, then it is described by a regular expression ($\Rightarrow$).

Equivalence with finite automata

Lemma

If a language is described by a regular expression, then it is regular ($\Leftarrow$).

Proof idea: Let's say that we have a regular expression R describing some language A. We show how to convert R into an *NFA* recognizing A. And we know that if a *NFA* recognize A then A is regular

Equivalence with finite automata

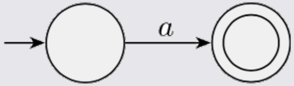
Lemma

If a language is described by a regular expression, then it is regular ($\Leftarrow$).

Proof.

Let us convert R into a *NFA N*. We do this by considering the six cases in the formal definition of regular expressions.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$ (language recognized by R). The below NFA recognizes $L(R)$

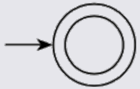


Equivalence with finite automata

Proof.

Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

2. $R = \epsilon$. Then $L(R) = \{\epsilon\}$. The below NFA recognizes $L(R)$



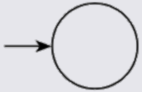
Formally, $N = \{\{q_1\}, \Sigma, \delta, q_1, \{q_1\}\}$, where $\delta(r, b) = \emptyset$ for any r and b

□

Equivalence with finite automata

Proof.

3. $R = \emptyset$. Then $L(R) = \emptyset$ so the following NFA recognizes $L(R)$



4. $R = R_1 \cup R_2$
5. $R = R_1 \circ R_2$
6. $R = R_1^*$

For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. We construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in the last case) and the appropriate closure construction. □

Examples

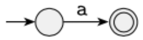
That ends the first part of the proof of theorem, giving the easier direction of the if and only if condition. Before going on to the other direction, let's consider some examples whereby we use this procedure to convert a regular expression to an NFA.

Examples

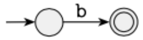
- Let's convert the regular expression $(ab \cup a)^*$ to a *NFA*
- We build up from the smallest subexpressions to larger subexpressions until we have an NFA for the original expression

Example: $(ab \cup a)^*$

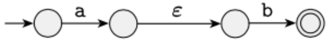
a



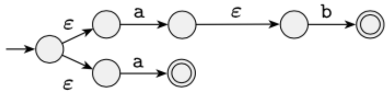
b



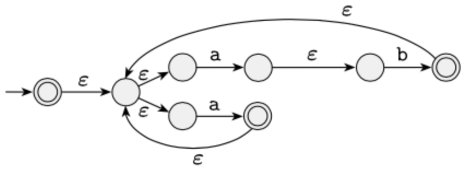
ab



ab ∪ a



(ab ∪ a)*



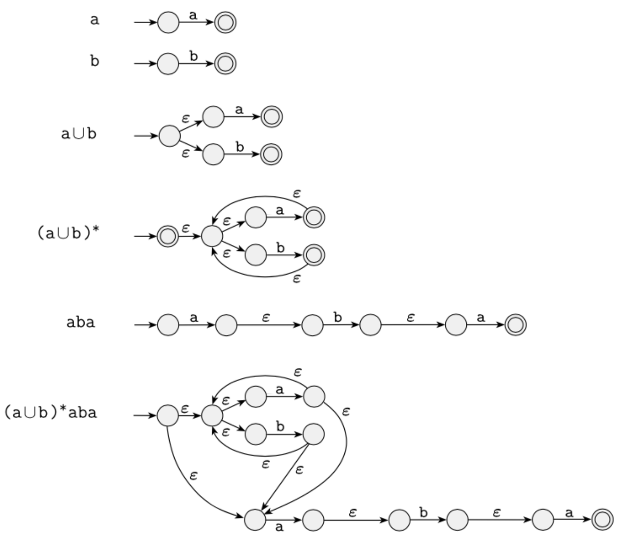
Example: $(ab \cup a)^*$

- Note that this procedure generally doesn't give the NFA with the fewest states.
- In this example, the procedure gives an NFA with eight states, but the smallest equivalent NFA has only two states. Can you find it? ($\Leftarrow$ first to find it will get a CS112 T-shirt)

Examples

- Let's convert the regular expression $(a \cup b)^*aba$ to a NFA
- We skip few minor steps

Example: $(a \cup b)^* aba$



Equivalence with finite automata

Lemma

If a language is regular, then it is described by a regular expression (⇒).

Proof idea:

- We need to show that if a language A is regular, a regular expression describes it. Because A is regular, it is accepted by a DFA. We describe a procedure for converting DFAs into equivalent regular expressions.
- We break this procedure into two parts, using a new type of finite automaton called a **generalized nondeterministic finite automaton (GNFA)**. First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Generalized Nondeterministic Finite Automaton

Generalized nondeterministic finite automata are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ϵ

- The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA
- The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow
- A GNFA is nondeterministic and so may have several different ways to process the same input string
- It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input

Example

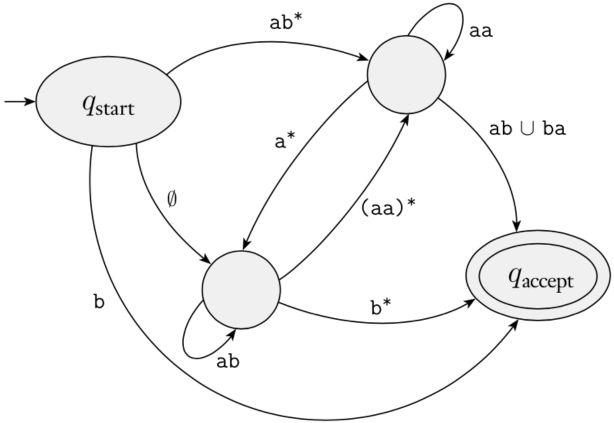


Figure: A generalized nondeterministic finite automaton

Generalized Nondeterministic Finite Automaton

We restrict our GNFAs to a special form that meets the following conditions:

- The start state has transition arrows going to every other state but no arrows coming in from any other state
- There is only a single accept state and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself

Convert DFA to GNFA

We can easily convert a DFA into a GNFA in the special form:

- We simply add a new start state with an ϵ arrow to the old start state and a new accept state with ϵ arrows from the old accept states
- If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels
- Finally, we add arrows labeled $\emptyset$ between states that had no arrows. This last step won't change the language recognized because a transition labeled with $\emptyset$ can never be used
- From here on we assume that all GNFAs are in the special form

Convert GNFA to RegExp

We study how to convert a GNFA into a regular expression:

- Let's assume that the GNFA has k states
- Because a GNFA must have a start and an accept state and they must be different, we know that $k \geq 2$
- If $k > 2$, we construct an equivalent GNFA with $k - 1$ states
- This step can be repeated on the new GNFA until it is reduced to two states
- If $k = 2$, then the GNFA as a single arrow that goes from the start state to the accept state
- The label of this arrow is the equivalent regular expression

DFA to RegExp

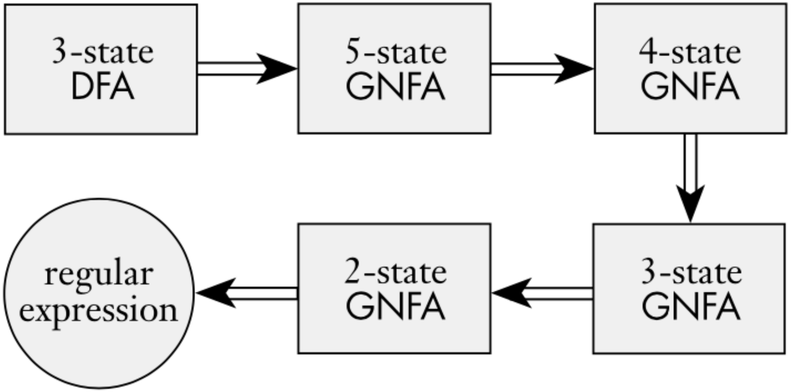


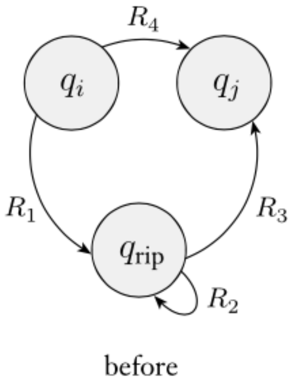
Figure: Typical stages in converting a DFA to a regular expression

DFA to RegExp

The crucial step is constructing an equivalent GNFA with one fewer state when $k > 2$

- We do so by selecting a state, ripping it out of the machine, and repairing the remainder so that the same language is still recognized
- We can use any state provided that it is not the start or accept state
- We are guaranteed that such a state will exist because $k > 2$
- We will call the removed state q_{rip} After removing q_{rip} we repair the machine by altering the regular expressions that label each of the remaining arrows
- The new labels compensate for the absence of qrip by adding back the lost computations
- The new label going from a state q_i to a state q_j is a regular expression that describes all strings that would take the machine from q_i to q_j either directly or thru q_{rip}

DFA to RegExp



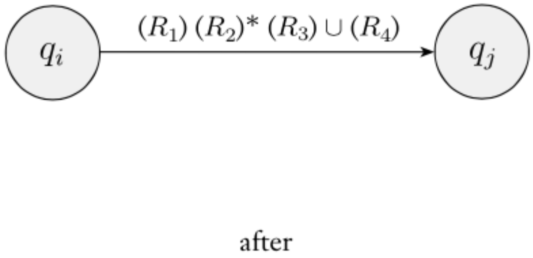
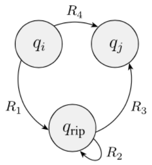
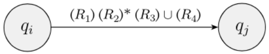


Figure: Constructing an equivalent GNFA with one fewer state

DFA to RegExp



before



after

In the initial machine:

- q_i goes tp q_{rip} witn an arrow labeled R_1
- q_{rip} goes to itself with an arrow labeled R_2
- q_{rip} goes to q_j with an arrow labeled R_3
- q_i goes to q_j with an arrow labeled R_4

Then in the new machine, we label q_i to q_j arrow as:

$$(R_1)(R_2)^*(R_3) \cup (R_4)$$

DFA to RegExp

We make this change for each arrow going from any state q_i to any state q_j, including the case where $q_i = q_j$. The new machine recognizes the original language.

This ends our proof idea, Now, we can move to formally describe it

Formal definition

Definition

A generalized nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$ where

1. Q is a finite set of states
2. Σ is the input alphabet
3. $\delta : (Q - \{q_{accept}\} \times (Q - \{q_{start}\})) \rightarrow \mathcal{R}$ is the transition function
4. q_{start} is the start state
5. q_{accept} is the accept state

GNFA Computation

Now we formalise GNFA computation as follows: Let $G = (Q, \Sigma, \delta, q_{start}, q_{accept})$ be a GNFA and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of Σ^*.

Definition

Then G **accepts** w if a sequence of states $q_0, q_1, \dots, q_k$ in Q exists with three conditions:

1. $q_0 = q_{start}$
2. $q_k = q_{accept}$
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$, meaning that R_i is the expression on the arrow from q_{i-1} to q_i

Equivalence with finite automata

Lemma

If a language is regular, then it is described by a regular expression ($\Rightarrow$).

Proof.

- Let M be the DFA for language A. Then we convert M to a GNFA G by adding a new start state and a new accept state and additional transition arrows as necessary
- We use the procedure $CONVERT(G)$, which takes a GNFA and returns an equivalent regular expression
- This procedure uses recursion:

Equivalence with finite automata

Proof.

```
CONVERT(G)
  1. Let $k$ be the number of states of $G$
  2. if $k = 2$ then $G$ must consist of a start state, an accept state, and a single arrow
     connecting them and labeled with a regular expression $R$
     Return the expression $R$
```

Equivalence with finite automata

Proof.

3. if $k > 2$ we select a state $q_{rip} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{start}, q_{accept})$ where:

$$Q' = Q - \{q_{rip}\}$$

   and for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$ let

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$$

   for $R_1 = \delta(q_i, q_{rip})$, $R_2 = \delta(q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$
4. Compute $CONVERT(G')$ and return this value

□

Equivalence with finite automata

Claim

For any GFA G, $CONVERT(G)$ is equivalent to G

Proof.

We prove this claim by induction on k, the number of states of the GNFA

Basic case

- Case $k = 2$. If G has only two states, it can have only a single arrow, which goes from the start state to the accept state. The regular expression label on this arrow describes all the strings that allow G to get to the accept state. Hence this expression is equivalent to G

Equivalence with finite automata

Proof.

Induction step

- Assume that claim is true for $k-1$. First we show that G and G' recognize the same language. Suppose that G accepts an input w. Then in an accepting branch of the computation, G enters a sequence of states:

$$q_{start}, q_1, q_2, q_3, \ldots, q_{accept}$$

- If none of them is the removed state q_{rip}, clearly G' also accepts w. The reason is that each of the new regular expressions labeling the arrows of G' contains the old regular expression as part of a union.

Equivalence with finite automata

Proof.

- If q_{rip} does appear, removing each run of consecutive q_{rip} states forms an accepting computation for G'. The states q_i and q_j bracketing a run have a new regular expression on the arrow between them that describes all strings taking q_i to q_j via q_{rip} on G. So G' accepts w.
- Conversely, suppose that G' accepts an input w. As each arrow between any two states q_i and q_j in G' describes the collection of strings taking q_i to q_j in G, either directly or via q_{rip}, G must also accept w. Thus G and G' are equivalent.
- The induction hypothesis states that when the algorithm calls itself recursively on input G', the result is a regular expression that is equivalent to G' because G' has $k-1$ states.

$\square$

Equivalence with finite automata

The above claim proves $\Rightarrow$ lemma. We already proved $\Leftarrow$ so we are done proving the theorem.

Example

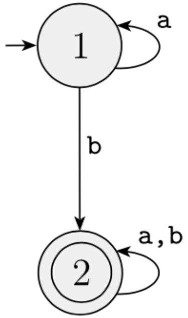


Figure: The NFA N_1

Let us take a two-state DFA.

Example

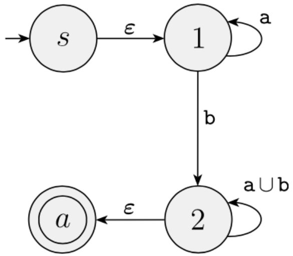


Figure: The NFA N_1

- We make a four-state GNFA by adding a new start state and a new accept state, called s and a for convenience
- We replace a, b on the self loop at state 2 with $a \cup b$

Example

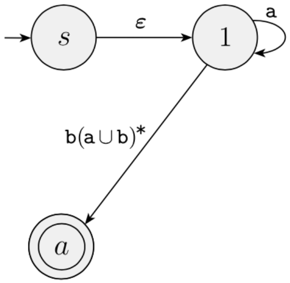


Figure: The NFA N_1

- We remove state 2 and update the remaining arrow labels
- The only label that changes is the one from 1 to a at it becomes $b(a \cup b)^*$
- q_i is state 1, q_j is state a and q_{rip} is 2. So we have $R_1 = b$, $R_2 = a \cup b$, $R_3 = \epsilon$ and $R_4 = \emptyset$.

Example

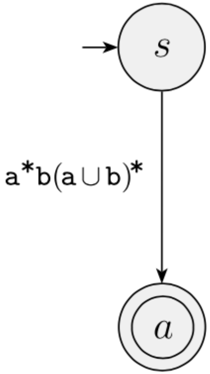


Figure: The NFA N_1

- We remove state 1 and follow the same procedure
- Because only the start and accept states remain, the label on the arrow joining them is the regular expression that is equivalent to the original DFA

Section 4

Practical RegEx

Practical RegEx

- Next slides gives a basic introduction to regular expressions themselves sufficient as conversational level and shows how regular expressions work in Python
- Based on Google for Education - Python

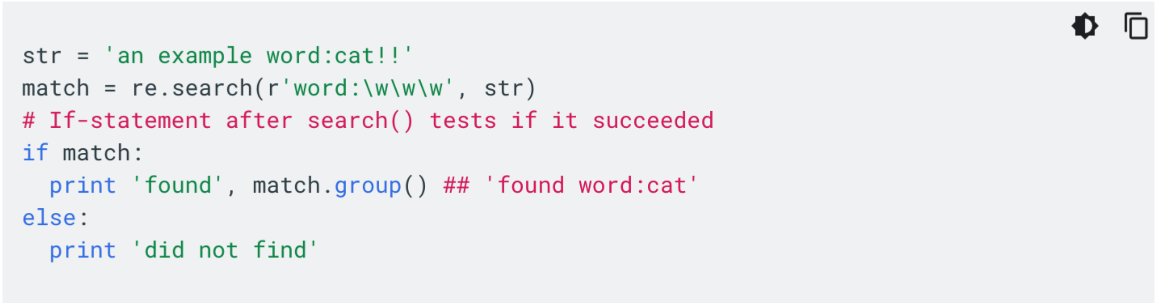
Practical RegEx

In Python a regular expression search is typically written as:

```
match = re.search(pat, str)
```

- The *re.search()* method takes a regular expression pattern and a string and searches for that pattern within the string
- If the search is successful, *search()* returns a match object or None otherwise
- Therefore, the search is usually immediately followed by an if-statement to test if the search succeeded

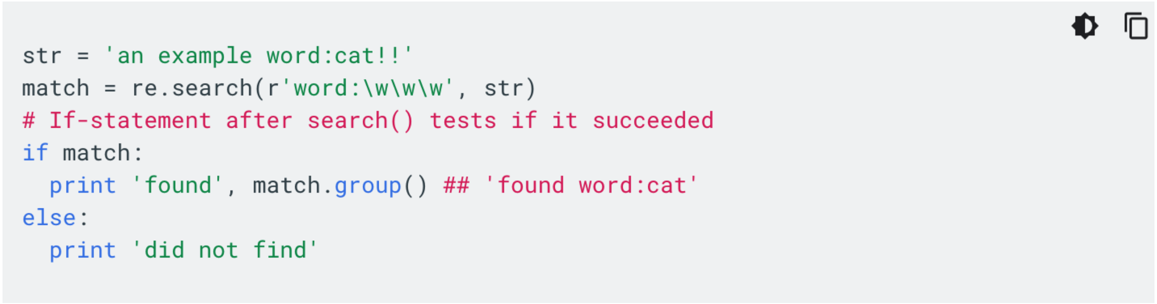
Practical RegEx



```
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)
# If-statement after search() tests if it succeeded
if match:
  print 'found', match.group() ## 'found word:cat'
else:
  print 'did not find'
```

- The code *match = re.search(pat, str)* stores the search result in a variable named "match"
- Then the if-statement tests the match – if true the search succeeded and match.group() is the matching text (e.g. 'word:cat')

Practical RegEx



```python
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)
# If-statement after search() tests if it succeeded
if match:
  print 'found', match.group() ## 'found word:cat'
else:
  print 'did not find'
```

- Otherwise if the match is false (None to be more specific), then the search did not succeed, and there is no matching text
- The 'r' at the start of the pattern string designates a python "raw" string which passes through backslashes without change which is very handy for regular expressions

Basic Patterns

The power of regular expressions is that they can specify patterns, not just fixed characters. Most basic patterns which match single chars:

- a, X, 9 – this are ordinary characters and just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: $, ^, ?, *, +, {, [,], (,), }, |
- . (a period) matches any single character except newline '\n'
- \w (lowercase w) – matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character
- \b – boundary between word and non-word
- \s – (lowercase s) matches a single whitespace character – space, newline, return, tab, form [\n \r \t \f]. \S (upper case S) matches any non-whitespace character

Basic Patterns

- \t, \n, \r – tab, newline, return
- \d – decimal digit [0-9] (some older regex utilities do not support \d, but they all support \w and \s)
- ˆ= start, $ = end – match the start or end of the string
- \– inhibit the "specialness" of a character. So, for example, use . to match a period or \\to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, , to make sure it is treated just as a character

Basic Example

The basic rules of regular expression search for a pattern within a string are:

- The search proceeds through the string from start to end, stopping at the first match found
- All of the pattern must be matched, but not all of the string
- If *match* = *re.search(pat, str)* is successful, match is not *None* and in particular *match.group()* is the matching text

Basic Example

```
## Search for pattern 'iii' in string 'piiig'.
## All of the pattern must match, but it may appear anywhere.
## On success, match.group() is matched text.
match = re.search(r'iii', 'piiig') # found, match.group() == "iii"
match = re.search(r'igs', 'piiig') # not found, match == None

## . = any char but \n
match = re.search(r'..g', 'piiig') # found, match.group() == "iig"

## \d = digit char, \w = word char
match = re.search(r'\d\d\d', 'p123g') # found, match.group() == "123"
match = re.search(r'\w\w\w', '@@abcd!!') # found, match.group() == "abc"
```

Repetition

Things get more interesting when you use + and * to specify repetition in the pattern:

- + – 1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
- * – 0 or more occurrences of the pattern to its left
- ? – match 0 or 1 occurrences of the pattern to its left

Leftmost & Largest

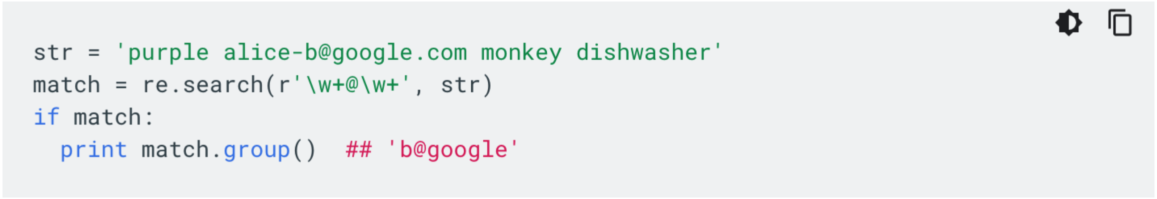
First the search finds the leftmost match for the pattern, and second it tries to use up as much of the string as possible – i.e. $+$ and * go as far as possible (the $+$ and * are said to be "greedy").

Repetition Examples

```
## i+ = one or more i's, as many as possible.
match = re.search(r'pi+', 'piiig') # found, match.group() == "piii"

## Finds the first/leftmost solution, and within it drives the +
## as far as possible (aka 'leftmost and largest').
## In this example, note that it does not get to the second set of i's.
match = re.search(r'i+', 'piigiiii') # found, match.group() == "ii"

## \s* = zero or more whitespace chars
## Here look for 3 digits, possibly separated by whitespace.
match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx') # found, match.group() == "1 2   3"
match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx') # found, match.group() == "12  3"
match = re.search(r'\d\s*\d\s*\d', 'xx123xx') # found, match.group() == "123"

## ^ = matches the start of string, so this fails:
match = re.search(r'^b\w+', 'foobar') # not found, match == None
## but without the ^ it succeeds:
match = re.search(r'b\w+', 'foobar') # found, match.group() == "bar"
```

Emails Example

Suppose you want to find the email address inside the string 'xyz alice-b@google.com purple monkey'. We'll use this as a running example to demonstrate more regular expression features. Here's an attempt using the pattern r'\w+@\w+':



```
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'\w+@\w+', str)
if match:
  print match.group()  ## 'b@google'
```

The search does not get the whole email address in this case because the \w does not match the '-' or '.' in the address. We'll fix this using the regular expression features on the next slide.

Square Brackets

Square brackets can be used to indicate a set of chars, so [abc] matches 'a' or 'b' or 'c'. The codes \w, \s etc. work inside square brackets too with the one exception that dot (.) just means a literal dot. For the emails problem, the square brackets are an easy way to add '.' and '-' to the set of chars which can appear around the @ with the pattern r'[\w.-]+@[\w.-]+' to get the whole email address:

```
match = re.search(r'[\w.-]+@[\w.-]+', str)
if match:
  print match.group()  ## 'alice-b@google.com'
```

You can also use a dash to indicate a range, so [a-z] matches all lowercase letters. To use a dash without indicating a range, put the dash last, e.g. [abc-]. An up-hat (^) at the start of a square-bracket set inverts it, so [^ab] means any char except 'a' or 'b'.

Construct a regular expression to detect URLs in a text file ($\Leftarrow$ first 3 will get a CS112 T-shirt)