SENG 265 - Fall 2017

Regular Expressions II

Bill Bird

Department of Computer Science
University of Victoria

November 8, 2017

More Regular Expressions in Python (1)

```
def is_match(s):
  if re.match('\w+ [Ll]ives',s):
    return True
  else:
    return False

>>> is_match('Elvis')
False
>>> is_match('Elvis Lives')
True
>>> is_match('Elvis Presley lives')
False
>>> is_match('Elvis Aaron Presley Lives')
False
```

- **Example**: Design a pattern to match strings of the form '`<Name> lives`' or '`<Name> Lives`', where name is at least one word (but may be multiple words).
- The above pattern is incorrect, since all of the last three strings should match.

More Regular Expressions in Python (2)

```
def is_match(s):
  if re.match('\w+ [Ll]ives',s):
    return True
  else:
    return False

>>> is_match('Elvis')
False
>>> is_match('Elvis Lives')
True
>>> is_match('Elvis Presley lives')
False
>>> is_match('Elvis Aaron Presley Lives')
False
```

- The pattern uses '\w+' to match a word. The pattern '[A-Za-z]+' would also work. For a proper name, '[A-Z][a-z]*' would probably be the most accurate.
- The problem with the pattern is that only one word of the name is matched.

More Regular Expressions in Python (3)

```
def is_match(s):
  if re.match('\w+ \w+ [Ll]ives',s):
    return True
  else:
    return False

>>> is_match('Elvis')
False
>>> is_match('Elvis Lives')
False
>>> is_match('Elvis Presley lives')
True
>>> is_match('Elvis Aaron Presley Lives')
False
```

- This version matches two words, but fails on the one word case.
- The lowercase 'l' in `lives` is successfully matched.

More Regular Expressions in Python (4)

```
def is_match(s):
  if re.match('\w+ \w+ \w+ [Ll]ives',s):
    return True
  else:
    return False

>>> is_match('Elvis')
False
>>> is_match('Elvis Lives')
False
>>> is_match('Elvis Presley lives')
False
>>> is_match('Elvis Aaron Presley lives')
True
```

- This version matches three word names, but fails on the one word case.
- The goal is to allow an unlimited number of words in the name. In the pattern, this corresponds to an unlimited number of '`\w+ `' terms.

More Regular Expressions in Python (5)

```
def is_match(s):
  if re.match('(\w+ )+[Ll]ives',s):
    return True
  else:
    return False

>>> is_match('Elvis')
False
>>> is_match('Elvis Lives')
True
>>> is_match('Elvis Presley lives')
True
>>> is_match('Elvis Aaron Presley lives')
True
```

- Using brackets around '`\w+ `' creates a subexpression, which can then be repeated one or more times with the + metasymbol.
- This version of the pattern is correct.

More Regular Expressions in Python (6)

```
def is_match(s):
  if re.match('(\w+ )+is dead',s):
    return True
  else:
    return False

>>> is_match('is dead')
False
>>> is_match('Bela Lugosi is dead')
True
>>> is_match('Bela Lugosi is undead')
False
>>> is_match('Bela is undead')
False
```

- **Example**: Design a pattern to match strings of the form '`<Name> is dead`' or '`<Name> is undead`', where name is again allowed to contain multiple words.

More Regular Expressions in Python (7)

```
def is_match(s):
  if re.match('(\w+ )+is dead',s):
    return True
  else:
    return False

>>> is_match('is dead')
False
>>> is_match('Bela Lugosi is dead')
True
>>> is_match('Bela Lugosi is undead')
False
>>> is_match('Bela is undead')
False
```

- Using the pattern from the previous example, it is easy to match the 'is dead' case.
- The last two strings should also match, though, so the pattern is incorrect.

More Regular Expressions in Python (8)

```
def is_match(s):
  if re.match('(\w+ )+is (dead|undead)',s):
    return True
  else:
    return False

>>> is_match('is dead')
False
>>> is_match('Bela Lugosi is dead')
True
>>> is_match('Bela Lugosi is undead')
True
>>> is_match('Bela is undead')
True
```

- Using the | ("or") metasymbol, the pattern can be modified to match either 'is dead' or 'is undead'.
- This pattern is correct.

More Regular Expressions in Python (9)

```
def is_match(s):
  if re.match('(\w+ )+is (un)?dead',s):
    return True
  else:
    return False

>>> is_match('is dead')
False
>>> is_match('Bela Lugosi is dead')
True
>>> is_match('Bela Lugosi is undead')
True
>>> is_match('Bela is undead')
True
```

- Another option is to use the ? operator to make the 'un' in 'undead' optional.
- This pattern is also correct.

Aside: Raw Strings (1)

Contrived Exercise: Design a regular expression to match a Windows drive name, such as '`C:\`', '`c:\`' or '`Z:\`'

- The drive letter can be any single uppercase or lowercase letter (Windows filenames are not case sensitive), so the pattern '`[A-Za-z]`' can be used to match it.
- The colon is required.
- The trailing backslash can be matched by the pattern '`\\`'.

The finished pattern is then

```
[A-Za-z]:\\
```

Aside: Raw Strings (2)

```
>>> import re

>>> drive_name = 'c:\\'
>>> print(drive_name)
c:\
>>> re.match('[A-Za-z]:\\',drive_name)
Traceback (most recent call last):
  File "/usr/lib/python3.2/sre_parse.py", line 194, in
      __next
 (Horrifying series of exceptions omitted)
sre_constants.error: bogus escape (end of line)
>>>
```

- Trying to match the pattern on the previous slide with `re.match` causes a parsing error.
- This is the result of both python and the `re` module trying to escape backslashes. Often, this behavior does not cause a parsing error as it does above, but prevents the pattern from working as planned.

Aside: Raw Strings (3)

```
>>> import re

>>> drive_name = 'c:\\'
>>> print(drive_name)
c:\
>>> pattern = '[A-Za-z]:\\'
>>> print(pattern)
[A-Za-z]:\
>>> re.match(pattern,drive_name)
Traceback (most recent call last):
 ...
>>>
```

- When the pattern is specified as a python string, the double backslash is escaped to a single backslash.
- The `re.match` function then receives the pattern '`[A-Za-z]:\`', which is invalid.

Aside: Raw Strings (4)

```
>>> import re

>>> drive_name = 'c:\\'
>>> print(drive_name)
c:\
>>> pattern = '[A-Za-z]:\\\\'
>>> print(pattern)
[A-Za-z]:\\
>>> re.match(pattern,drive_name)
<_sre.SRE_Match object at 0x1c89100>
>>>
```

- One solution is to double escape all backslashes. This is not ideal.
- Consider a pattern for full Windows paths such as 'H:\seng265\a1\ass1.c'

Aside: Raw Strings (5)

```
>>> import re

>>> drive_name = 'c:\\'
>>> print(drive_name)
c:\
>>> pattern = r'[A-Za-z]:\\'
>>> print(pattern)
[A-Za-z]:\\
>>> m = re.match(pattern,drive_name)
>>> print(m)
<_sre.SRE_Match object at 0x1c89100>
```

- Python's *raw string* feature disables escape characters during string parsing.
- Prefixing a string constant with `r` enables raw string mode.

Aside: Raw Strings (6)

```
>>> import re

>>> S1 = 'These\tare\\escape\ncharacters\"'
>>> print(S1)
These	are\escape
characters"
>>> S2 = r'These\tare\\escape\ncharacters\"'
>>> print(S2)
These\tare\\escape\ncharacters\"
>>>
```

- Raw strings are helpful for cases where a string constant should contain literal escape sequences like '`\n`', and are especially useful for regular expressions.
- It is considered good style to use raw strings for all regular expressions in Python.

Aside: Raw Strings (7)

```
>>> import re

>>> S1 = r'H:\seng265\a1\ass1.c'
>>> print(S1)
H:\seng265\a1\ass1.c
>>> S2 = r'H:\'
  File "<stdin>", line 1
    S2 = r'H:\'
              ^
SyntaxError:  EOL while scanning string literal
```

- All backslashes inside a raw string are retained as entered.
- **Bizarre Exception**: Raw strings cannot end with a backslash (so the earlier drive name '`C:\`' cannot be entered as a raw string).
- The Python developers do not have a good explanation for this exception, and the documentation contains a very weak excuse for it.

Example: Matching C declarations (1)

Example: Design a regular expression to match a C variable declaration with base type `int`. Declarations of arrays with any number of dimensions should also be matched. For this example, only arrays with a specified size are allowed (so '`int A[];`' should not match). The table below gives examples of strings which should and should not match.

Should Match	**Should Not Match**
`int x;`	`float f;`
`int var1, var2;`	`int 3var;`
`int A[10];`	`int A[10]`
`int A[10][20],B[10];`	`int [10][20];`
`int A[10][20],B[10],x;`	`int;`

Example: Matching C declarations (2)

```
def is_match(s):
  if re.match(r'int [A-Za-z_]\w*;',s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int x')
False
>>> is_match('int     x;')
False
>>> is_match('int var1;')
True
>>> is_match('int ;')
False
```

- Since regular expressions are cryptic, it is often best to develop them iteratively starting from a simple case.
- The above pattern seems to work correctly when a single variable is being declared, except when excess whitespace is present.

Example: Matching C declarations (3)

```
def is_match(s):
  if re.match(r'int\s*[A-Za-z_]\w*;',s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int x')
False
>>> is_match('int     x;')
True
>>> is_match('intx;')
True
```

- Using '\s*' matches any amount of whitespace.
- However, using '*' allows 0 whitespace characters, which is incorrect.

Example: Matching C declarations (4)

```
def is_match(s):
  if re.match(r'int\s+[A-Za-z_]\w*;',s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int x')
False
>>> is_match('int     x;')
True
>>> is_match('intx;')
False
```

- Changing the pattern to use '`\s+`' requires at least one space after '`int`'.

Example: Matching C declarations (5)

```
def is_match(s):
  if re.match(r'int\s+[A-Za-z_]\w*(,\s*[A-Za-z_]\w*)*;',s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int     x;')
True
>>> is_match('int var1, var2;')
True
>>> is_match('int var1,     var2;')
True
>>> is_match('int var1,;')
False
>>> is_match('int ,var2;')
False
```

- To allow multiple variables to be declared, the subexpression '`,\s*[A-Za-z_]\w*`' is repeated zero or more times after the first variable name.
- The first variable name is left out of the subexpression to force at least one variable name to be present.

Example: Matching C declarations (6)

```
def is_match(s):
  if re.match(r'int\s+[A-Za-z_]\w*(,\s*[A-Za-z_]\w*)*;',s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int x,y,    z;')
True
>>> is_match('int A[10];')
False
>>> is_match('int A[10][20];')
False
```

- The pattern from the previous slide matches declaration of one or more primitive variables, but it does not match any array declarations.
- To adapt the pattern for arrays, it is probably better to start by separately designing a pattern to match a single array declaration (since the pattern above is very complicated already).

Example: Matching C declarations (7)

```
def is_match(s):
  if re.match(r'int\s+[A-Za-z_]\w*;',s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int x,y,    z;')
False
>>> is_match('int A[10];')
False
>>> is_match('int A[10][20];')
False
```

- To design the array version, we start with the single variable version from earlier.
- An array declaration, such as '`int A[10][20];`' consists of a name (in this case '`A`'), followed by zero or more `[size]` specifiers. When zero `[size]` specifiers are present, a regular variable is being declared.

Example: Matching C declarations (8)

```
def is_match(s):
  if re.match(r'int\s+[A-Za-z_]\w*(\[\d+\])*;',s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int x,y,    z;')
False
>>> is_match('int A[10];')
True
>>> is_match('int A[10][20];')
True
>>> is_match('int A[abc];')
False
```

- The pattern '`\[\d+\]`' will match one `[size]` specifier (note the escaped square brackets).
- Therefore, '`(\[\d+\])*`' will match zero or more `[size]` terms.

Example: Matching C declarations (9)

```
pattern = \ #Line split to allow pattern to fit the page
r'int\s+[A-Za-z_]\w*(\[\d+\])*(,\s*[A-Za-z_]\w*(\[\d+\])*)*;'
def is_match(s):
  if re.match(pattern,s):
    return True
  return False
>>> is_match('int x;')
True
>>> is_match('int x,y,    z;')
True
>>> is_match('int A[10];')
True
>>> is_match('int A[10][20];')
True
>>> is_match('int x,      y,A[10];')
True
>>> is_match('int x, A[10],y;')
True
```

- Finally, the '`(\[\d+\])*`' terms are added to the multi-variable pattern to produce the finished pattern.
- Editorial remark: The finished pattern is **ugly**.

Complexity of Regular Expressions

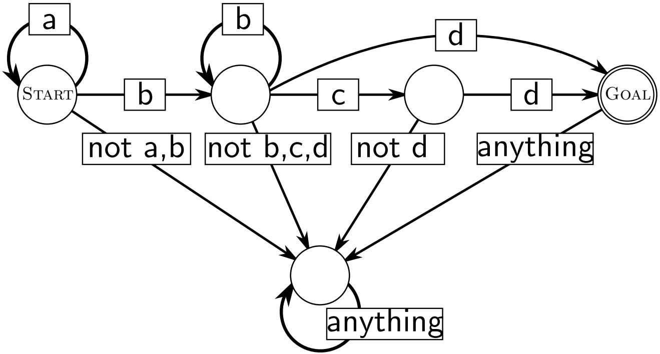
Parsing and Matching

- Directly matching a string of length n against a pattern of length m requires $O(nm)$ time.
- The running time might be greater if certain non-standard features are used.

Pre-parsing

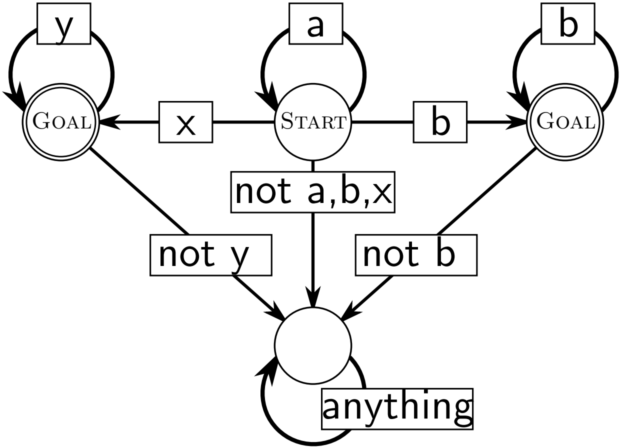
- Every regular expression is equivalent to a deterministic finite automaton (DFA).
- Matching a string using a DFA requires O(n) time.
- Pre-parsing a regular expression and converting it to a DFA requires $O(2^m)$ time, but is a better choice for matching large numbers of strings against the same pattern.

Regular Expressions and DFAs (1)



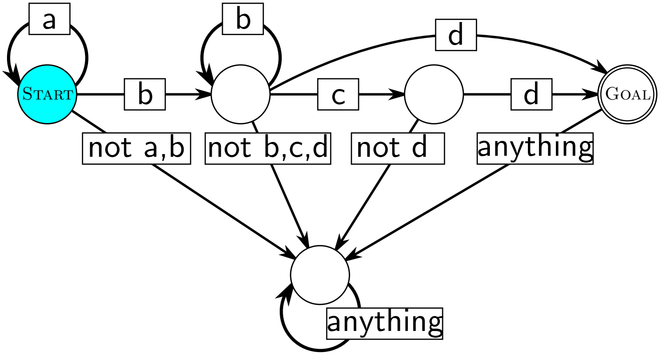
- A DFA is a form of state machine, and can be represented by a graph with a vertex for each state.
- The above DFA is equivalent to the regular expression '`a*b+c?d`'

Regular Expressions and DFAs (2)



- The above DFA is equivalent to the regular expression 'a*(b+|xy*)'.

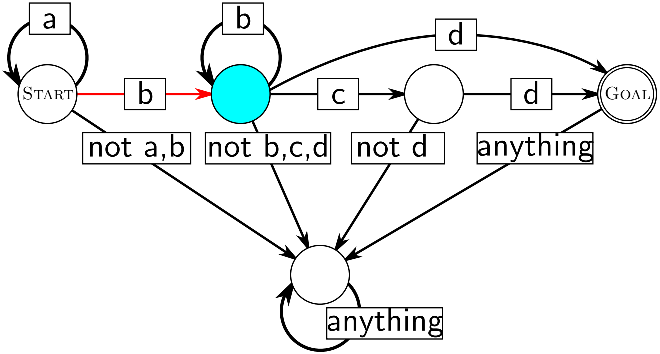
Regular Expressions and DFAs (3)



S = ‘`bbcd`’

- To test whether a string is a match, begin at the START state.

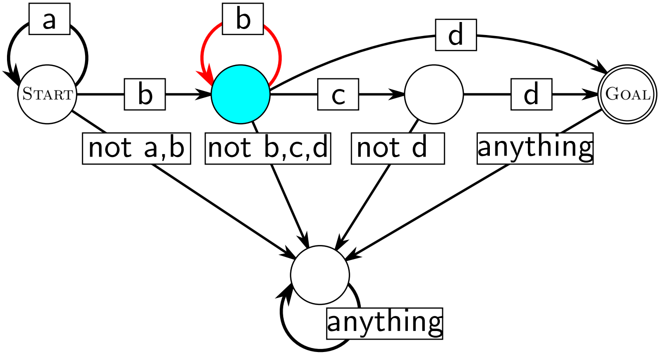
Regular Expressions and DFAs (4)



$S =$ '`bbcd`'

- At each step, move along the edge corresponding to the next character of the string.

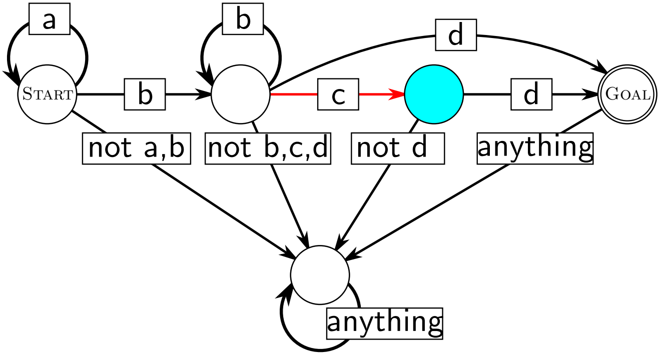
Regular Expressions and DFAs (5)



$S =$ '`bbcd`'

- At each step, move along the edge corresponding to the next character of the string.

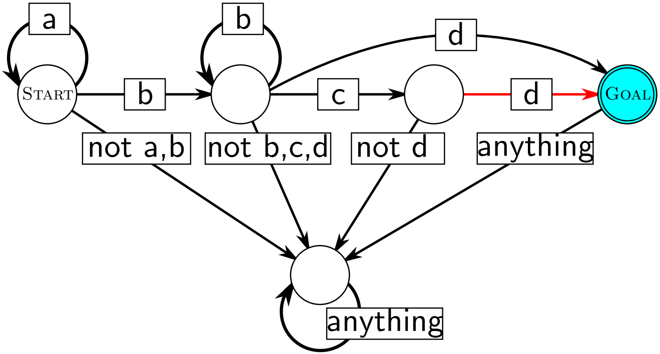
Regular Expressions and DFAs (6)



$S =$ '**bbc**d'

- At each step, move along the edge corresponding to the next character of the string.

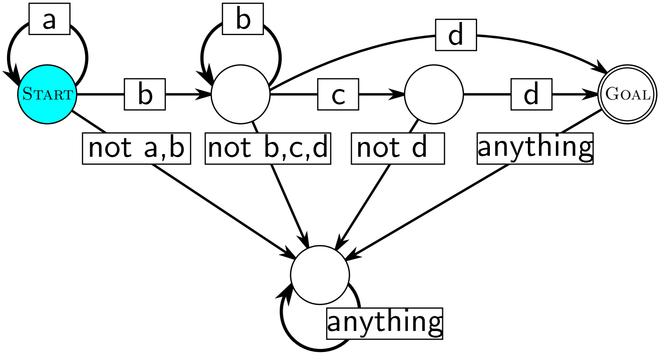
Regular Expressions and DFAs (7)



$S =$ '`bbcd`'

- If, at the end of the string, the current state is a goal state, the string is a match.
- The above string is a match for '`a*b+c?d`'.

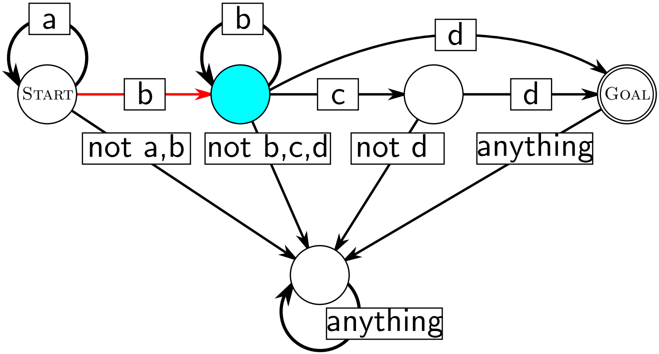
Regular Expressions and DFAs (8)



$S = $ ‘bb’

- A string with n characters can be processed in $O(n)$ time with a DFA.

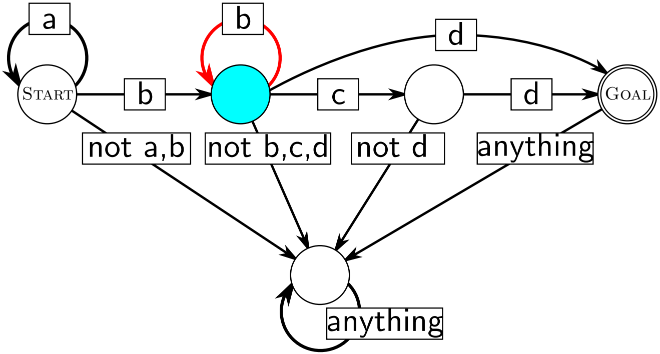
Regular Expressions and DFAs (9)



$S =$ '**b**b'

- A string with n characters can be processed in $O(n)$ time with a DFA.

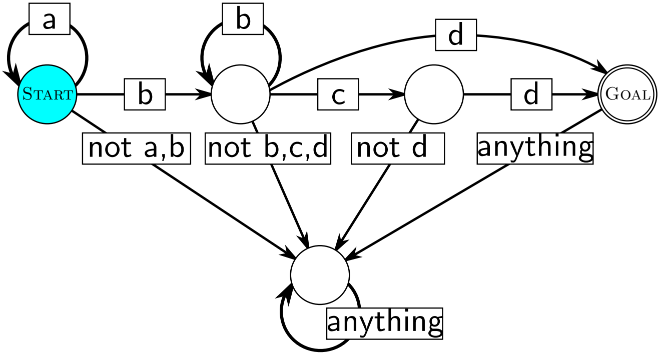
Regular Expressions and DFAs (10)



$S =$ '`bb`'

- The traversal for '`bb`' ends at a non-goal state, so the string is not a match.
- A DFA may contain multiple goal states.

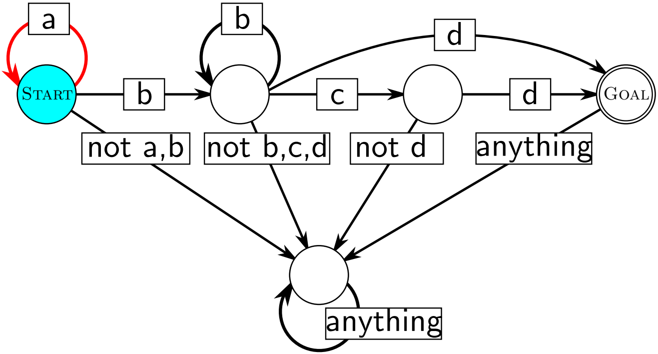
Regular Expressions and DFAs (11)



$S =$ '`aabab`'

- For complicated patterns, the DFA can have size exponential in the length of the regular expression.

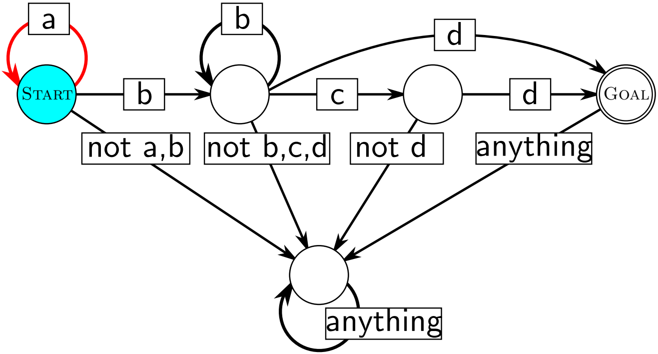
Regular Expressions and DFAs (12)



$S =$ '`aabab`'

- Converting a regular expression of length m to a DFA may require $O(2^m)$ time.

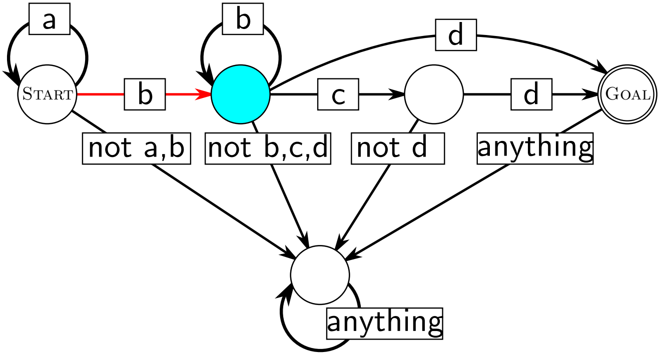
Regular Expressions and DFAs (13)



S = ‘aabab’

- The specifics of the conversion are covered in CSC 320.

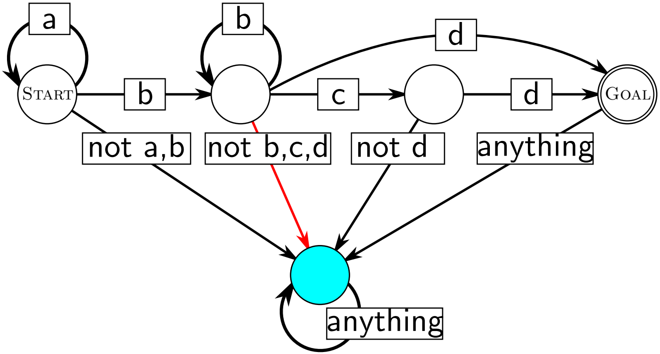
Regular Expressions and DFAs (14)



$S =$ 'aabab'

- DFAs and other automata are also significant topics in CSC 435.

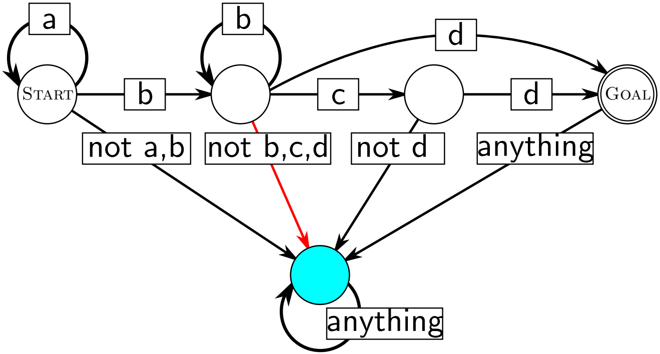
Regular Expressions and DFAs (15)



$S =$ '**aaba**b'

- The state at the bottom is a 'dead state', since it is not a goal state and there is no transition away from it.

Regular Expressions and DFAs (16)



$S =$ '`aabab`'

- When the traversal eventually finishes, it will still be stuck in the dead state.
- Since the final state is not a goal state, the string is not a match.

Compiled Patterns (1)

```
>>> m = re.match('^a*(b*|c*)$', 'aaab')
>>> print(m)
<_sre.SRE_Match object at 0x1c9b5d0>
>>> m = re.match('^a*(b*|c*)$', 'aaabc')
>>> print(m)
None

>>> pattern = re.compile('^a*(b*|c*)$')
>>> m = pattern.match('aaab')
>>> print(m)
<_sre.SRE_Match object at 0x1c9b5d0>
>>> m = pattern.match('aaabc')
>>> print(m)
None
```

- If a pattern will be used repeatedly, it can be pre-compiled with the `re.compile` method into a pattern object.
- Using a precompiled pattern eliminates most of the overhead of parsing the pattern.

Compiled Patterns (2)

```
>>> m = re.match('^a*(b*|c*)$', 'aaab')
>>> print(m)
<_sre.SRE_Match object at 0x1c9b5d0>
>>> m = re.match('^a*(b*|c*)$', 'aaabc')
>>> print(m)
None

>>> pattern = re.compile('^a*(b*|c*)$')
>>> m = pattern.match('aaab')
>>> print(m)
<_sre.SRE_Match object at 0x1c9b5d0>
>>> m = pattern.match('aaabc')
>>> print(m)
None
```

- **Erroneous Claim**: "Repeatedly using a pattern without pre-compiling will require the `re` module to repeatedly parse the pattern".

Compiled Patterns (3)

```
>>> m = re.match('^a*(b*|c*)$', 'aaab')
>>> print(m)
<_sre.SRE_Match object at 0x1c9b5d0>
>>> m = re.match('^a*(b*|c*)$', 'aaabc')
>>> print(m)
None

>>> pattern = re.compile('^a*(b*|c*)$')
>>> m = pattern.match('aaab')
>>> print(m)
<_sre.SRE_Match object at 0x1c9b5d0>
>>> m = pattern.match('aaabc')
>>> print(m)
None
```

- The `re` module keeps an internal cache of previously used patterns, so repeated uses of the same pattern will not usually require re-parsing.
- It is still considered good style to pre-compile commonly-used patterns.

Extracting Data (1)

```
>>> S = 'Latitude 48.465, Longitude 236.686 - Overcast'
>>> P1 = 'Latitude [\d.]*, Longitude [\d.]* - \w*'
>>> m = re.match(P1,S)
>>> print(m)
<_sre.SRE_Match object at 0x7fb6faca3850>
>>> print(m.groups())
()

>>> P2 = 'Latitude ([\d.]*), Longitude ([\d.]*) - (\w*)'
>>> m = re.match(P2,S)
>>> print(m)
<_sre.SRE_Match object at 0x7fb6fabb5ca8>
>>> print(m.groups())
('48.465', '236.686', 'Overcast')
```

- The match object returned by `re.match` can be used to extract information from the matched text.
- The ( ) operator has two functions in a regular expression: creating subexpressions and capturing data.

Extracting Data (2)

```
>>> S = 'Latitude 48.465, Longitude 236.686 - Overcast'
>>> P1 = 'Latitude [\d.]*, Longitude [\d.]* - \w*'
>>> m = re.match(P1,S)
>>> print(m)
<_sre.SRE_Match object at 0x7fb6faca3850>
>>> print(m.groups())
()

>>> P2 = 'Latitude ([\d.]*), Longitude ([\d.]*) - (\w*)'
>>> m = re.match(P2,S)
>>> print(m)
<_sre.SRE_Match object at 0x7fb6fabb5ca8>
>>> print(m.groups())
('48.465', '236.686', 'Overcast')
```

- Each pair of parentheses creates a ‘group’ in the pattern, and when the pattern is matched, all text inside the parentheses is associated with the group.

Extracting Data (3)

```
>>> S = '<td>48.465</td><td>236.686</td><td>Overcast</td>'
>>> P1 =
    '<td>([\w.]*)</td><td>([\w.]*)</td><td>([\w.]*)</td>'
>>> m = re.match(P1,S)
>>> print(m)
<_sre.SRE_Match object at 0x7fb6fabb5ca8>
>>> print(m.groups())
('48.465', '236.686', 'Overcast')

>>> P2 = '^(<td>([\w.]*)</td>){3}$'
>>> m = re.match(P2,S)
>>> print(m)
<_sre.SRE_Match object at 0x1d17b58>
>>> print(m.groups())
('<td>Overcast</td>', 'Overcast')
```

- The number of groups available in the final match is equal to the number of pairs of parentheses.
- In the second example above, the subexpression '`(<td>([\w.]*)</td>)`' is matched three times, but only the last match is stored in the group.

Extracting Data (4)

$$\texttt{\^{}}\overbrace{\texttt{(<td>}\underbrace{\texttt{([\textbackslash w.]*)}}_{\text{group 2}}\texttt{</td>)}}^{\text{group 1}}\texttt{\{3\}\$}$$

- Groups are numbered (starting at 1) based on the order of their left brackets.
- The contents of a group can be matched inside a regular expression with specifiers '\1', '\2', '\3', ...

Extracting Data (5)

```
>>> S = '<td>48.465</td><td>236.686</td><td>Overcast</td>'
>>> P1 = '<td>[\w.]*</td>'
>>> re.findall(P1,S)
['<td>48.465</td>', '<td>236.686</td>', '<td>Overcast</td>']

>>> P2 = '<td>([\w.]*)</td>'
>>> re.findall(P2,S)
['48.465', '236.686', 'Overcast']
```

- The `re.findall` function is useful for extracting all matches from a string.
- Without groups (as in the top example), the return value of `re.findall` is a list of the non-overlapping instances of the pattern in the string.

Extracting Data (6)

```
>>> S = '<td>48.465</td><td>236.686</td><td>Overcast</td>'
>>> P1 = '<td>[\w.]*</td>'
>>> re.findall(P1,S)
['<td>48.465</td>', '<td>236.686</td>', '<td>Overcast</td>']

>>> P2 = '<td>([\w.]*)</td>'
>>> re.findall(P2,S)
['48.465', '236.686', 'Overcast']
```

- Without using groups, it is difficult to extract only the desired data from the string.
- When groups are present, the return value of `re.findall` is a list of the groups associated with each non-overlapping occurrence of the pattern.

Extracting Data (7)

```
S = """
Latitude 48.465, Longitude 236.686 - Overcast
Latitude 48.461, Longitude -123.311 - Rain
Latitude 40.133, Longitude -105.282 - Flurries
Latitude 50.725, Longitude 15.608 - Fair
"""
>>> P1 = 'Latitude [\d.-]*, Longitude [\d.-]* - \w*'
>>> re.findall(P1,S)
['Latitude 48.465, Longitude 236.686 - Overcast',
 'Latitude 48.461, Longitude -123.311 - Rain',
 'Latitude 40.133, Longitude -105.282 - Flurries',
 'Latitude 50.725, Longitude 15.608 - Fair']
>>> P2 = 'Latitude ([\d.-]*), Longitude ([\d.-]*) - (\w*)'
>>> re.findall(P2,S)
[('48.465', '236.686', 'Overcast'),
 ('48.461', '-123.311', 'Rain'),
 ('40.133', '-105.282', 'Flurries'),
 ('50.725', '15.608', 'Fair')]
```

- When a pattern contains multiple groups, the list returned by `re.findall` contains a tuple for each occurrence of the pattern.

Non-capturing groups

```
>>> S = 'The rain in Spain stays mainly in the plain.'
>>> P1 = '\w*ain\w*'
>>> re.findall(P1,S)
['rain', 'Spain', 'mainly', 'plain']
>>> P2 = '(r|pl)ain\w*'
>>> re.findall(P2,S)
['r', 'pl']
>>> P3 = '(?:r|pl)ain\w*'
>>> re.findall(P3,S)
['rain', 'plain']
```

- The pattern `P2` above uses brackets to create a subexpression, without the intention of capturing a group.
- Since a group is present, `re.findall` only returns the group contents, not the entire match.
- Using a 'non-capturing' group, as in pattern `P3`, resolves this problem.

Substitution

```
>>> S = "Gregor Samsa, R. Nigel Horspool, Bill Bird"
>>> P1 = r'[A-Z][a-z]+ [A-Z][a-z]+'
>>> re.findall(P1,S)
['Gregor Samsa', 'Nigel Horspool', 'Bill Bird']

>>> re.sub(P1,'Name',S)
'Name, R. Name, Name'

>>> P2 = r'([A-Z])([a-z]+) ([A-Z][a-z]+)'
>>> re.findall(P2,S)
[('G', 'regor', 'Samsa'), ('N', 'igel', 'Horspool'),
 ('B', 'ill', 'Bird')]

>>> re.sub(P2,r'\1. \3',S)
'G. Samsa, R. N. Horspool, B. Bird'
```

- The function `re.sub(pattern,substitution,S)` replaces all occurrences of `pattern` in `S` with `substitution`.
- The substitution string can reference groups captured by the matched pattern.

Lookahead and Lookbehind (1)

```
>>> S = "Gregor Samsa, R. Nigel Horspool, Bill Bird"

>>> P1 = '(?<=[A-Z])[a-z]+(?= [A-Z][a-z]+)'
>>> re.findall(P1,S)
['regor', 'igel', 'ill']

>>> re.sub(P1,'.',S)
'G. Samsa, R. N. Horspool, B. Bird'
```

- Often, a regular expression is used to isolate a set of characters appearing in a certain context.
- For example, extracting all sequences of digits which appear between a capital letter and a lowercase letter (such as ‘A1234b’).
- The context is important for finding the sequence, but not useful for data extraction.

Lookahead and Lookbehind (2)

```
>>> S = "Gregor Samsa, R. Nigel Horspool, Bill Bird"

>>> P1 = '(?<=[A-Z])[a-z]+(?= [A-Z][a-z]+)'
>>> re.findall(P1,S)
['regor', 'igel', 'ill']

>>> re.sub(P1,'.',S)
'G. Samsa, R. N. Horspool, B. Bird'
```

- Python offers 'lookaround' expressions, which match against surrounding characters without consuming those characters as part of the match.
- 'Lookbehind' expressions, of the form '`(?<=...)`', match leading characters, and 'lookahead' expressions, of the form '`(?=...)`' match trailing characters.
- The same effect can often be achieved with careful use of groups.

Lookahead and Lookbehind (3)

```
>>> S = "Gregor Samsa, R. Nigel Horspool, Bill Bird, Elvis
    Aaron Presley"
>>> P1 = '([A-Z])([a-z]+) ([A-Z][a-z]+)'
>>> re.findall(P1,S)
[('G', 'regor', 'Samsa'), ('N', 'igel', 'Horspool'),
 ('B', 'ill', 'Bird'), ('E', 'lvis', 'Aaron')]
>>> re.sub(P1,r'\1. \3',S)
'G. Samsa, R. N. Horspool, B. Bird, E. Aaron Presley'

>>> P2 = '(?<=[A-Z])[a-z]+(?= [A-Z][a-z]+)'
>>> re.findall(P2,S)
['regor', 'igel', 'ill', 'lvis', 'aron']
>>> re.sub(P2,'.',S)
'G. Samsa, R. N. Horspool, B. Bird, E. A. Presley'
```

- Using groups to filter out context information can result in some instances not being matched.
- The goal of pattern `P1` is to abbreviate all first names to initials.
- However, '`Elvis Aaron Presley`' is only abbreviated to '`E. Aaron Presley`'.

Lookahead and Lookbehind (4)

```
>>> S = "Gregor Samsa, R. Nigel Horspool, Bill Bird, Elvis
    Aaron Presley"
>>> P1 = '([A-Z])([a-z]+) ([A-Z][a-z]+)'
>>> re.findall(P1,S)
[('G', 'regor', 'Samsa'), ('N', 'igel', 'Horspool'),
 ('B', 'ill', 'Bird'), ('E', 'lvis', 'Aaron')]
>>> re.sub(P1,r'\1. \3',S)
'G. Samsa, R. N. Horspool, B. Bird, E. Aaron Presley'

>>> P2 = '(?<=[A-Z])[a-z]+(?= [A-Z][a-z]+)'
>>> re.findall(P2,S)
['regor', 'igel', 'ill', 'lvis', 'aron']
>>> re.sub(P2,'.',S)
'G. Samsa, R. N. Horspool, B. Bird, E. A. Presley'
```

- The matches returned for `P1` do not include the tuple `('A', 'aron', 'Presley')`.
- This is because the match overlaps with the previous match, `('E','lvis','Aaron')`.
- Lookaround expressions do not consume any characters, so no overlapping occurs.

Example: Matching C Comments (1)

Example: Design a pattern to match a single C comment (of the '`/* */`' form). Use the pattern and `re.findall` to extract the contents of every C comment in a given string of text.

Strings like '`/**/`', '`/* A comment */`' and '`/*int x;*/`' should match the pattern.

Strings like '`int x; /* A variable */`' and '`/* A comment */ int x; /* Another comment */`' should not match.

Example: Matching C Comments (2)

```
def is_match(s):
  if re.match(r'^/\*.*\*/$',s):
    return True
  return False
>>> is_match('/* */')
True
>>> is_match('/* A comment */')
True
>>> is_match('/* int x; */')
True
>>> is_match('int x; /* A variable */')
False
>>> is_match('/* A comment */ int x; /* Another comment */')
True
```

- A C comment consists of the characters '/*' followed by any text, followed by the characters '*/'.
- A simple pattern to try is '^/*.**/$'

Example: Matching C Comments (3)

```
def is_match(s):
  if re.match(r'^/\*.*\*/$',s):
    return True
  return False
>>> is_match('/* */')
True
>>> is_match('/* A comment */')
True
>>> is_match('/* int x; */')
True
>>> is_match('int x; /* A variable */')
False
>>> is_match('/* A comment */ int x; /* Another comment */')
True
```

- The highlighted string containing two comments does match the description on the previous slide.
- The pattern must prevent the characters '*/' from appearing inside the comment.

Example: Matching C Comments (4)

```
def is_match(s):
  if re.match(r'^/\*([^*]|\*[^/])*\*/$',s):
    return True
  return False
>>> is_match('/* */')
True
>>> is_match('/* A comment */')
True
>>> is_match('/* int x; */')
True
>>> is_match('int x; /* A variable */')
False
>>> is_match('/* A comment */ int x; /* Another comment */')
False
```

- Instead of using '`.*`', to match the inside of the comment, the pattern '`([^*]|*[^/])*`' can be used instead.

Example: Matching C Comments (5)

```
>>> S = '/* A comment */ int x; /* another comment */'
>>> P1 = '/\*.*\*/'
>>> re.findall(P1,S)
['/* A comment */ int x; /* another comment */']

>>> P2 = '/\*([^*]|\*[^/])*\*/'
>>> re.findall(P2,S)
[' ', ' ']

>>> P3 = '/\*(?:[^*]|\*[^/])*\*/'
>>> re.findall(P3,S)
['/* A comment */', '/* another comment */']

>>> P4 = '/\*((?:[^*]|\*[^/])*)\*/'
>>> re.findall(P4,S)
[' A comment ', ' Another comment ']
```

- Using the original pattern to extract all comments has the predictable effect.

Example: Matching C Comments (6)

```
>>> S = '/* A comment */ int x; /* another comment */'
>>> P1 = '/\*.*\*/'
>>> re.findall(P1,S)
['/* A comment */ int x; /* another comment */']

>>> P2 = '/\*([^*]|\*[^/])*\*/'
>>> re.findall(P2,S)
[' ', ' ']

>>> P3 = '/\*(?:[^*]|\*[^/])*\*/'
>>> re.findall(P3,S)
['/* A comment */', '/* another comment */']

>>> P4 = '/\*((?:[^*]|\*[^/])*)\*/'
>>> re.findall(P4,S)
[' A comment ', ' Another comment ']
```

- Using the improved pattern results in two matches being found by `re.findall`, but the captured text is wrong.
- This is caused by the brackets in the improved pattern.

Example: Matching C Comments (7)

```
>>> S = '/* A comment */ int x; /* another comment */'
>>> P1 = '/\*.*\*/'
>>> re.findall(P1,S)
['/* A comment */ int x; /* another comment */']

>>> P2 = '/\*([^*]|\*[^/])*\*/'
>>> re.findall(P2,S)
[' ', ' ']

>>> P3 = '/\*(?:[^*]|\*[^/])*\*/'
>>> re.findall(P3,S)
['/* A comment */', '/* another comment */']

>>> P4 = '/\*((?:[^*]|\*[^/])*)\*/'
>>> re.findall(P4,S)
[' A comment ', ' Another comment ']
```

- Changing the brackets to be a non-capturing group fixes the problem.
- The extracted data consists of both comments in their entirety.

Example: Matching C Comments (8)

```
>>> S = '/* A comment */ int x; /* another comment */'
>>> P1 = '/\*.*\*/'
>>> re.findall(P1,S)
['/* A comment */ int x; /* another comment */']

>>> P2 = '/\*([^*]|\*[^/])*\*/'
>>> re.findall(P2,S)
[' ', ' ']

>>> P3 = '/\*(?:[^*]|\*[^/])*\*/'
>>> re.findall(P3,S)
['/* A comment */', '/* another comment */']

>>> P4 = '/\*((?:[^*]|\*[^/])*)\*/'
>>> re.findall(P4,S)
[' A comment ', ' Another comment ']
```

- To extract only the contents of the comments, a capturing group can be placed around the entire interior of the comment in the pattern.

Example: Matching C Comments (9)

```
>>> S = '/* A comment */ int x; /* another comment */'
>>> P5 = '/\*.*\*/'
>>> re.findall(P5,S)
['/* A comment */ int x; /* Another comment */']

>>> P6 = '/\*.*?\*/'
>>> re.findall(P6,S)
['/* A comment */', '/* Another comment */']

>>> P7 = '/\*(.*?)\*/'
>>> re.findall(P7,S)
[' A comment ', ' Another comment ']
```

- The simple pattern doesn't work because the '`.*`' sub-pattern is *greedy* and produces the longest possible match.
- In Python RE syntax, adding '?' after a quantifier (such as '*', '+' or '?') makes that quantifier non-greedy.
- The non-greedy '`.*`' does not consume the '*/' since the next element of the pattern is '*/'.