

Regular Expressions

Motivation

Handling white space

A program ought to be able to treat any number of white space characters as a separator

Identifying blank lines

Most people consider a line with just spaces on it to be blank

Parsing input from html files

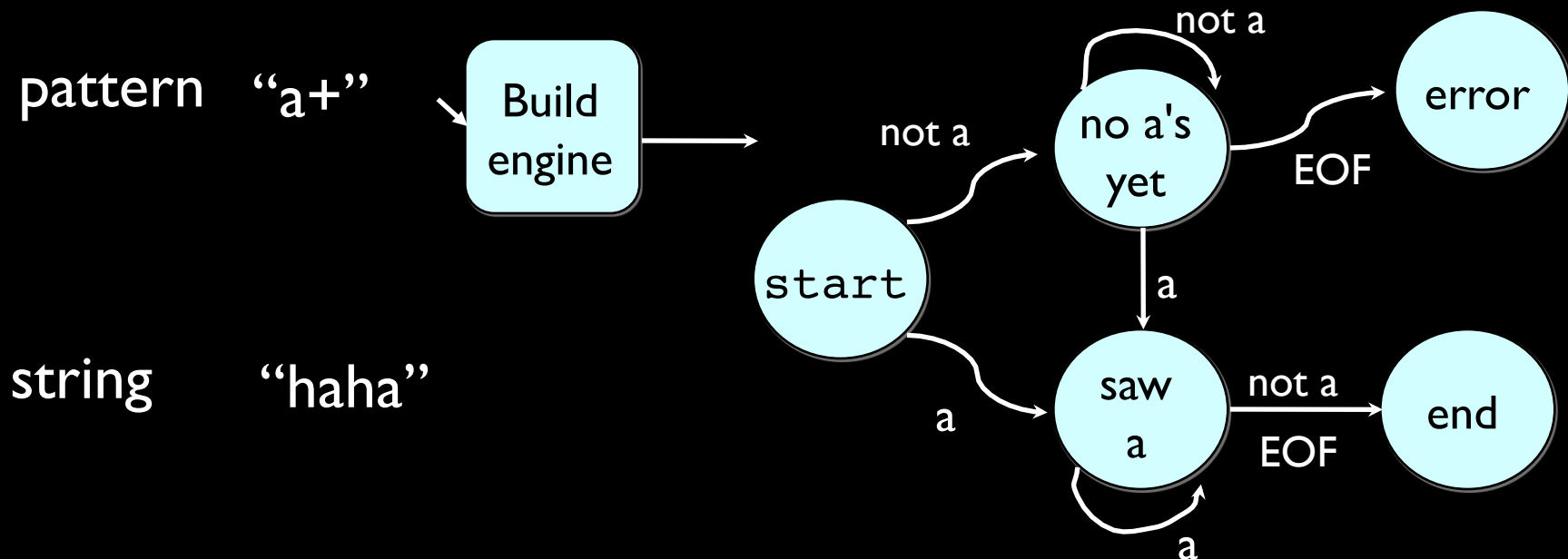
Searching for specifically formatted text in a large file, e.g. currency, date, etc.

Writing code to examine characters one by one is painful!

Regular Expressions

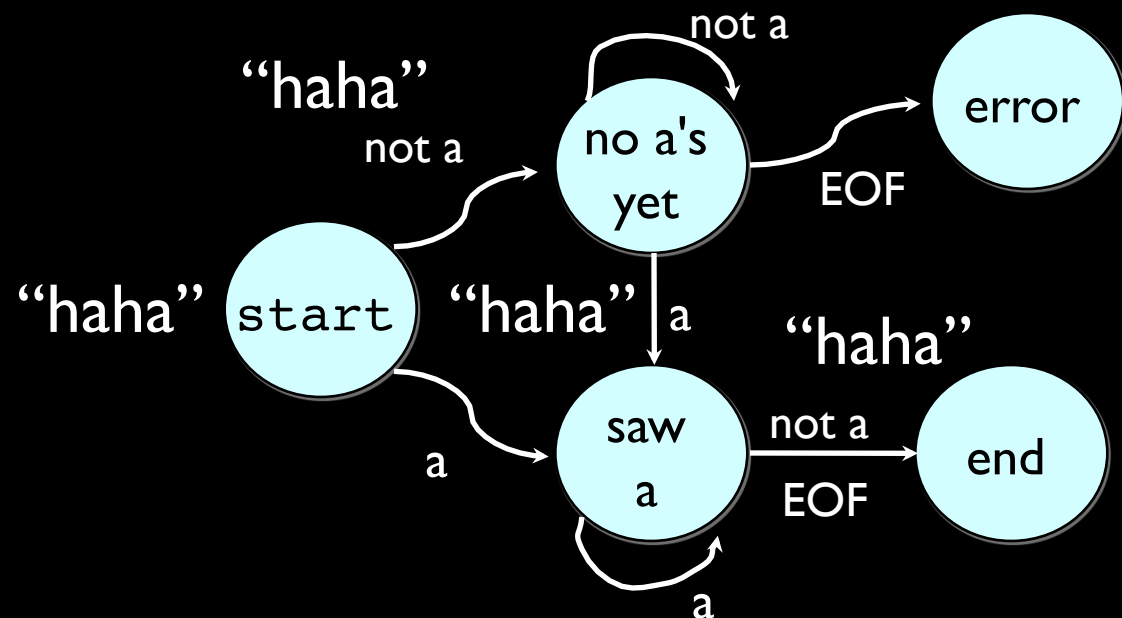
A way to represent patterns by strings

The matcher converts a pattern into a finite state machine and then compares a string to the state machine to find a match



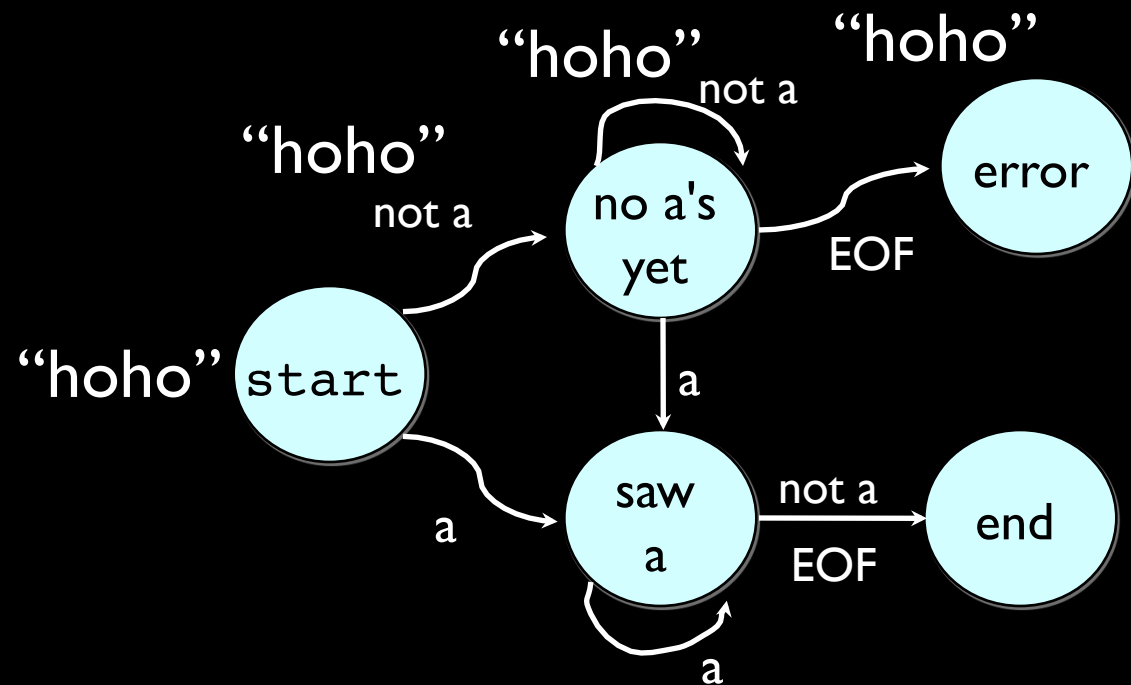
Matching a String

String "haha"



Matching a String

String "hoho"



Simple Patterns

| Pattern | Matches | Explanation |
|---------|---------------|----------------|
| a* | ", 'a', 'aa' | zero or more |
| b+ | 'b', 'bb' | one or more |
| ab?c | 'ac', 'abc' | one or zero |
| [abc] | 'a', 'b', 'c' | one from a set |
| [a-c] | 'a', 'b', 'c' | abbreviation |
| [abc]* | ", 'acbccb' | combination |

How To Use In Java

```
Scanner inputReader = new Scanner(System.in);
String line = inputReader.next();
String sampleRegex = "a[bc]*";
Pattern samplePattern = Pattern.compile(sampleRegex);
while (!line.equals("quit")) {
    Matcher mo = samplePattern.matcher(line);
    if (mo.find()) {
        System.out.println("MATCHED!\n");
    } else {
        System.out.println("NOT MATCHED!\n");
    }
    line = inputReader.next();
}
```

Anchoring

Force the position of match

`^` matches the beginning of the line

`$` matches the end

Neither consumes any characters.

| pattern | text | result |
|--------------------|-------|-----------------------|
| <code>b+</code> | abbc | Matches |
| <code>^b+</code> | abbc | Fails (no b at start) |
| <code>^a*\$</code> | aabaa | Fails (not all a's) |

Escaping

Match actual `^` and `$` using escape sequences `\^` and `\$`.

Match actual `+` and `*` using escape sequences `\+` and `*`.

Be careful with back slashes.

Use escapes for other characters:

`\t` is a tab character.

`\n` is a newline.

Look in the *API* for a full list of pattern options and characters that can be escaped.

Example: Counting Blank Lines

Want to find this pattern:

start of line, any number of spaces, tabs, carriage
returns, and newlines, end of line

```
Scanner fileContents = new Scanner(new File(fileName));
String blank = "^[ \\t\\n\\r]*$";
Pattern blankPattern = Pattern.compile(blank);
int count = 0;
while (fileContents.hasNext()) {
    Matcher mo = blankPattern.matcher(fileContents.next());
    if (mo.find())
        count++;
}
System.out.println(count);
```

Character sets

Use escape sequences for common character sets

| | | |
|-----------------|-----------------------------|---------------------------|
| <code>\d</code> | Digits | <code>[0-9]</code> |
| <code>\w</code> | Word | <code>[a-zA-Z0-9_]</code> |
| <code>\s</code> | Space | <code>[\t\n\r]</code> |
| <code>.</code> | Anything except end of line | <code>[^\n]</code> |

The notation `[^abc]` means “anything not in the set”

Match Objects

The `Matcher` object returned by `samplePattern.matcher()` has some useful methods:

`mo.group()` returns the string that matched.

`mo.start()` and `mo.end()` are the match's location.

Example:

```
Pattern samplePattern = Pattern.compile("b+");
```

```
Matcher mo = samplePattern.matcher("abbcb");
```

```
System.out.println(mo.group() + " " + mo.start() + " " + mo.end());
```

Sub-Matches

All parenthesized sub-patterns are remembered.

Text that matched Nth parentheses (counting from left) is group N.

```
String numbered = "\\s*(\\d+)\\s*:";
Pattern numberedPattern = Pattern.compile(numbered);
Matcher mo = numberedPattern.matcher(
    "Part 1: foo, Part 2: bar");
while (mo.find()) {
    String num = mo.group(1);
    System.out.println(num);
}
```

Example

```
String nameCase = "[^A-Z]*([A-Z][a-z]*) (.*)";
Pattern nameCaseP = Pattern.compile(nameCase);
while(inputReader.hasNextLine()) {
    Matcher mo = nameCaseP.matcher(line);
    while(mo.find()) {
        System.out.println(mo.group(1));
        mo = nameCaseP.matcher(mo.group(2));
    }
}
```

input:

```
This is a sample document. It has several words in name
case on the same line. It was written in August of
2003.
```

Advance patterns

Patterns:

a | b

ab | cd

a (bc | de) f

(\w) \1

a{2,3}

Matches:

'a', 'b'

'ab', 'cd'

'abcf', 'adef'

'aa', 'bb', 'zz', 'qq'

'aa', 'aaa'

Final Word

The methods and examples demonstrated in these notes are barely scratching the surface. Don't forget to look for more useful methods in the Java API. Classes to look up: `Pattern`, and `Matcher` from the `java.util.regex` library

Curious about Regular Expressions in Python? Have a look at the documentation for the `re` module.

Sample Questions

Full name, e.g. Foo Bar

9 digit student number, e.g. 123456789

Postal Code, e.g. M2N 7L6

Simple math formula: number operation number = ?

Date in the format mm/dd/yyyy, no need to worry about 28-day years or lengths of months. This is easy but longer than expected

Canadian Currency, e.g. CAD\$ 34.50, or CAD\$ 29

Imaginary filename format: 3 digits followed by 4 alphabetic characters followed by the exact same digits as the first part, followed by a “.imaginary” extension