Session Objectives

- Use regular expressions for pattern matching
- Define finite state machines
- Draw a diagram of a finite state machine
- Use a finite state machine to parse words

Pattern Matching with Regular Expressions

- The most common way to put regular expressions to work is in text processing.
- Some powerful utilities (such as `sed`, `grep` and `egrep`) support pattern matching using regular expressions to define the patterns.
- There are even some programming languages such as Perl, Tcl and Python which provide built-in support for such pattern matching.
- Perl is the language of choice for writing CGI scripts precisely because it makes it easy to split apart the complex strings that `POST` commands in HTML spit out.

Example Application

A tool that will check for ‘doubled words’ (such as “this this”).

- Accept any number of files, reporting each line of each file that has doubled words, with the source filename on each line.
- Work across lines, even finding doubled words where a word at the end of one line also occurs at the beginning of the next line.
- Find doubled words despite capitalisation differences, such as “The the” as well as allowing differing amounts of whitespace (spaces, tabs, new-lines etc.) between the words.
- Find doubled words that might be separated by HTML tags, such as `<B>very</B> very`

Friedl’s solution (in Perl) consists of three regular expressions with associated text substitutions (in a program of just six lines).

A Real Application

- While developing the Netlab neural network toolbox, I wrote the reference manual in LaTeX.
- I also needed to generate a reference manual in HTML and help text for Matlab, the programming language used for the toolbox.
- Clearly, what I didn't want to do was to type out all the text with different formatting commands for the different presentations. This would take a very long time, and whenever the reference material was updated, it would be necessary to update two other formats as well.
- LaTeX uses a mark-up language to change the presentation of text: these commands (such as `\emph` to emphasise text) can be matched and modified.
- I wrote two Perl scripts that could be run over all the Matlab programs and reference material to generate the different formats whenever I created a new release.

Pattern Matching with `egrep` and `grep`

- You can't learn the Perl language in a lecture, so we will not look at editing text, but just at the pattern matching process.
- A typical `egrep` command:

  ```
  egrep 'hello' *
  ```

- This searches for the string "hello" (which is the regular expression) in all files in the current directory.
- Note the use of quotes around the string. These are not part of the regular expression, but are needed by the command shell (the part of the system that accepts typed commands and executes the programs you specify).

-
  ```
  egrep 'MyInt' *.adb
  ```

  finds all occurrences of the type name `MyInt` in the Ada source files in the current directory.

Regular Expression Syntax in `egrep`

- The alphabet Σ is assumed to be all ASCII characters.
- Concatenation is defined by writing words next to each other.
- Kleene closure is defined by the `*` character.
- Union is denoted by $|$ with round brackets used as delimiters. For example, the regular expression `(m|b)ad` matches "mad" and "bad", while `(From|Subject|Date)` matches each of the given strings.

There are other metacharacters that `egrep` provides to make it easier to write compact regular expressions.

Metacharacters in `egrep`

Start/End of Line The caret `^` and dollar `$` represent the start and end of a line of text. So `^cat` matches a line with `cat` at the start. The caret anchors the regular expression to the start of the line. The expression `mat$` matches `mat` at the end of a line.

Character Classes The `[]` construct, known as a character class, lets you list the characters you want at a certain point in a regular expression. So `[Tt]his` matches both `This` and `this`. In this form, the construct is equivalent to a union of individual characters. Where such classes really come into their own is to define a range of characters. So `<H[1-6]>` matches all levels of HTML header, from `<H1>` to `<H6>`. The expression `[0-9a-fA-F]` is useful when matching the characters used in hexadecimal numbers.

The single dot character `.` is a shorthand for a character class that matches any character. So `.ar` matches `aar`, `2ar` etc.

Repetition

There are several useful operators that allow us to specify the number of times an expression should be repeated with precision.

`?`	Preceding item is matched at most once.
`*`	Preceding item is matched zero or more times.
`+`	Preceding item is matched one or more times.
`{n}`	Preceding item is matched exactly n times.
`{n,}`	Preceding item is matched n or more times.
`{n,p}`	Preceding item is matched at least n times but not more than p times.

Examples

HTML directives The HTML specification states that spaces are allowed immediately before the closing >, so `<H1>` and `<H2 >` are both legal tags. We can match all these variants in a single regular expression

```
<H[1-6] *>
```

Decimal numbers To match all legal natural numbers in base 10, we can use the expression

```
(0|[1-9][0-9]*)
```

This ensures that the number is either zero, or has a leading digit that is non-zero.

Word Boundaries The symbols `\<` and `\>` match the empty string at the beginning and end of a word (which is any sequence of alphanumeric characters). The expression `\<cat\>` matches `cat` but not `catalogue` or `scat`. The symbol `\b` matches the empty string at either beginning or end of a word.

Escaping Because some characters are actually metacharacters, they can't be used directly in regular expressions. For example, to search for text representing a sum of money in dollars, we would like to type `$[0-9]+.[0-9]{2}`, but the initial dollar sign matches an end of word, and the decimal point matches any character. We can make sure that we match these two problematic cases as characters by escaping them with a preceding backslash `\`. So

```
egrep '\$[0-9]+\.[0-9]{2}'
```

successfully matches strings like `$145.62`

Exercise

How would you modify the regular expression

```
egrep '\$[0-9]+\.[0-9]{2}'
```

so that whole numbers of dollars like `$10020` are also matched?

Solution

How would you modify the regular expression

```
egrep '\$[0-9]+\.[0-9]{2}'
```

so that whole numbers of dollars like `$10020` are also matched?

```
\$[0-9]+(\.[0-9]{2})?
```

Remembering Matches

- An extension to regular expressions that is supported by some versions of `egrep` that takes us slightly outside the regular languages but can still be parsed by variants on the standard parser (though potentially at the expense of a significantly slower running time).
- Parentheses can ‘remember’ text matched by the subexpression they enclose. To refer to a matched subexpression, the expression `\n` is used, where n is the index of the bracketed expression.
- For example,

  ```
  (a)(b)(c)\3\2\1
  ```

  matches the string `abccba`.

Example

- We can use this construct to give a partial solution to the doubled word problem mentioned at the start of this section.

```
egrep -i '\b([a-z]+)( +)(\1\b)' test.txt
```

- This command searches in a case-insensitive way (the `-i` flag) for a word (of at least one character) `\b([a-z]+)` followed by an arbitrary number of spaces `( +)` before matching the same word again `(\1\b)`.
- The `\b` metacharacters ensure that we don't match strings like `then the` or `this is`.
- The only important part of the problem that this does not solve is the case when the two word occurrences are separated by a line break.

When I ran this command on the current set of lecture notes, this was the output:

```
\emph{conjunctive normal form} (\emph{CNF}) it is is a conjunction
\emph{conjunctive normal form} (\emph{CNF}) it is is a conjunction
the \emph{scope} of the quantifier.  An occurrence of a a variable $x$
(such as ``this this''), a common problem in documents.  Your job is to
are called compiler compilers.  YACC (Yet Another Compiler Compiler),
wittily called Bison, generate C programs.  The Java Compiler Compiler,
```

The fourth of these refers to the definition of a repeated word, and the fifth and sixth refer to a type of program (called a compiler compiler), so are OK, but the other three are incorrect (and have now been fixed).

Automata and Parsing

- A parser for a regular expression can be implemented as a simulation of a simple type of abstract machine.
- A machine is an abstract computer that can read strings in a certain language and which tells us, after a finite number of steps, whether a given string belongs to the language.

Deterministic Finite Automata

- There is a finite set of internal states Q.
- There is a finite alphabet of symbols Σ that the machine can read.
- The machine has an initial state $q_0 \in Q$.
- The machine has a set of final (or accepting) states $F \subseteq Q$.
- There is a transition (or next-state) function $f : Q \times \Sigma \rightarrow \Sigma$ so that when the machine is in state q and reads character a, it moves to $f(q, a)$.

The machine is deterministic in the sense that it operation is completely determined by the input string. If the machine ends up in a final state, then it is said to accept the input; otherwise it rejects it.

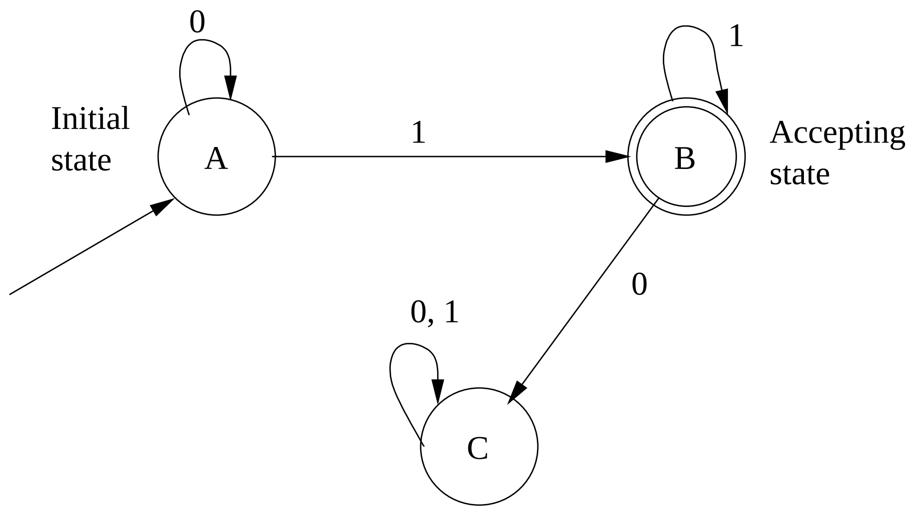
Properties of DFA

- DFA are extremely rudimentary machines. Once an input symbol has been read, it is lost. There is a sort of memory associated with the internal states, but it is severely limited.
- It is easy to see how a DFA can act as a parser for the language that it accepts.
- The key result for the application of DFAs is Kleene's theorem that a language is accepted by some DFA if and only if it is regular.
- The proof of this theorem is beyond the scope of this course, but it does give an algorithm for constructing a DFA that accepts a given regular language.

Drawing DFA

Consider a DFA with state set $\{A, B, C\}$, alphabet $\{0, 1\}$, initial state A, accepting state B and transition function:

Input	0	1
Present State		
A	A	B
B	C	B
C	C	C



Consider what happens when the string 00110 is read.

Example Continued

- Careful consideration of the transition function shows that the DFA will read an initial sequence of 0s of arbitrary length staying in state A.
- A single 1 will move it to B, where it will stay so long as it continues to read 1s.
- After the next zero, it moves to C, where it will then stay for ever. (Such a state is called a dead state).
- It is relatively easy to see that the regular expression `0*1*` defines the language accepted by this DFA.

Session Objectives

- Use regular expressions for pattern matching
- Define finite state machines
- Draw a diagram of a finite state machine
- Use a finite state machine to parse words