# Pattern matching and lexing

Informatics 2A: Lecture 6

Alex Simpson

School of Informatics
University of Edinburgh
als@inf.ed.ac.uk

26 September, 2014

## Recap of Lecture 5

- Regular languages are closed under the operations of concatenation and Kleene star.
- This is proved using $\epsilon$-NFAs, which can be easily converted to ordinary NFAs.
- Regular expressions provide a textual representation of regular languages.
- Kleene's Theorem: regular expressions define exactly the regular languages.
- The difficult direction of the theorem, that every regular language is defined by some regular expression, can be proved using Kleene algebra to solve a system of simultaneous equations, exploiting Arden's Rule.

## Applications of regular-language machinery (i.e. of finite automata and regular expressions)

Applications of regular expressions encountered in Inf1-DA:

- Specifying the structure of XML documents using DTDs (Document Type Definitions)
- Searching for concordances (and more general forms of pattern matching) in a corpus (e.g., using CQP).

In this lecture we consider two further applications:

- Pattern matching in UNIX/Linux/Mac OS X, using grep
- Lexing: the first stage in the formal-language-processing 'pipeline' introduced in Lecture 2 (Course Roadmap).

Another application to morphological parsing will be covered in Lecture 14.

## Pattern matching with Grep tools

Important practical problem: Search a large file (or batch of files) for strings of a certain form.

Most UNIX/Linux-style systems since the '70s have provided a bunch of utilities for this purpose, known as Grep (Global Regular Expression Print).

Extremely useful and powerful in the hands of a practised user. Make serious use of the theory of regular languages.

Typical uses:

```
grep "[0-9]*\.[0-9][0-9]" document.txt
```

−− searches for prices in pounds and pence

```
egrep "(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)" document.txt
```

−− searches for occurrences of the word "the"

## grep, egrep, fgrep

There are three related search commands, of increasing generality and correspondingly decreasing speed:

- `fgrep` searches for one or more fixed strings, using an efficient *string matching* algorithm.
- `grep` searches for strings matching a certain pattern (a simple kind of regular expression).
- `egrep` searches for strings matching an extended pattern (these give the full power of regular expressions).

For us, the last of these is the most interesting.

## Syntax of patterns (a selection)

| | |
|---|---|
| `a` | Single character |
| `[abc]` | Choice of characters |
| `[A-Z]` | Any character in ASCII range |
| `[^Ss]` | Any character except those given |
| `.` | Any single character |
| `^`, `$` | Beginning, end of line |
| `*` | zero or more occurrences of preceding pattern |
| `?` | optional occurrence of preceding pattern |
| `+` | one or more occurrences of preceding pattern |
| `\|` | choice between two patterns (‘union’) |

(N.B. The last three of these are specific to `egrep`.)

This kind of syntax is very widely used. In Perl/Python (including NLTK), patterns are delimited by `/.../` rather than `"..."`.

## Mathematical versus pattern syntax

Don't be confused by the mimatch between the mathematical syntax for regular expressions (as used, e.g., in Kleene algebra) and the pattern syntax for regular expressions.

The union of two languages is written using $+$ in mathematical syntax, and | in pattern syntax.

In pattern syntax, $+$ is a unary operation representing one or more concatenations of strings satisfying the pattern

| mathematical | pattern |
|---|---|
| $\alpha + \beta$ | $\alpha \mid \beta$ |
| $\alpha\alpha^*$ | $\alpha+$ |

## Self-assessment question

Which of the patterns below are suitable for matching non-negative decimal integers in a written English document?

1. `[0-9]*`
2. `[0-9]+`
3. `0 | [1-9][0-9]*`
4. `0 | [1-9][0-9]?[0-9]?(,[0-9][0-9][0-9])*`

## How egrep (typically) works

`egrep` will print all lines containing a match for the given pattern. How can it do this efficiently?

- Patterns are clearly regular expressions in disguise.
- So we can convert a pattern into a (smallish) NFA.

  (More precisely, the number of states of the NFA grows linearly in the length of the regular expression.)
- We then run the NFA , using the just-in-time simulation discussed in Lecture 4.

  We do not determinize the NFA to construct the full DFA, because of the potential exponential state-space blow-up.

`grep` can be a bit more efficient, exploiting the fact that there's 'less non-determinism' around in the absence of $+, ?, |$.

## Challenge question

Regular expressions and the pattern language have operations that correspond to the closure of regular languages under union, concatenation and Kleene star.

However, we have seen other closure properties of regular languages too: closure under intersection and complement.

Question: Why do regular expressions and patterns not include operations for intersection and complement?

## Lexical analysis of formal languages

Another application: lexical analysis (a.k.a. lexing).

The problem: Given a source text in some formal language, split it up into a stream of lexical tokens (or lexemes), each classified according to its lexical class.

Example: In Java,

```
while(count2<=1000)count2+=100
```

would be lexed as

| | | | | | |
|---|---|---|---|---|---|
| `while` | `(` | `count2` | `<=` | `1000` | `)` |
| WHILE | LBRACK | IDENT | INFIX-OP | INT-LIT | RBRACK |
| `count2` | `+=` | `100` | | | |
| IDENT | ASS-OP | INT-LIT | | | |

## Lexing in context

- The output of the lexing phase (a stream of tagged lexemes) serves as the input for the parsing phase.

  For parsing purposes, tokens like 100 and 1000 can be conveniently lumped together in the class of *integer literals*. Wherever 100 can legitimately appear in a Java program, so can 1000.

  Keywords of the language (like `while`) and other special symbols (like brackets) typically get a lexical class to themselves.

- Another job of the lexing phase is to throw away whitespace and comments.

  Rule of thumb: Lexeme boundaries are the places where a space could harmlessly be inserted.

## Lexical tokens and regular languages

In most computer language (e.g. Java), the allowable forms of identifiers, integer literals, floating point literals, comments etc. are simple enough to be described by regular expressions.

This means we can use the technology of finite automata to produce efficient lexers.

Even better, if you're designing a language, you don't actually need to write a lexer yourself!

Just write some regular expressions that define the various lexical classes, and let the machine automatically generate the code for your lexer.

This is the idea behind lexer generators, such as the UNIX-based `lex` and the more recent Java-based `jflex`.

## Sample code (from Jflex user guide)

```
Identifier = [:jletter:] [:jletterdigit:]*
DecIntegerLiteral = 0 | [1-9][0-9]*
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
EndOfLineComment     = "//" {InputCharacter}* {LineTerminator}
```

... and later on ...

```
{"while"}            { return symbol(sym.WHILE); }
{Identifier}         { return symbol(sym.IDENT); }
{DecIntegerLiteral}  { return symbol(sym.INT_LIT); }
{"=="}               { return symbol(sym.ASS_OP); }
{EndOfLineComment}   { }
```

## Nerd question

A correct pattern defining `jletterdigit` is:

```
[0-9] | [a-z] | [A-Z]
```

What is wrong with the pattern below?

```
[0-9] | [A-z]
```

1. It does not make sense.
2. It matches the symbol '['.
3. It does not match any letter.
4. No idea.

## Recognizing a lexical token using NFAs

- Build NFAs for our lexical classes $L_1, \ldots, L_k$ in the order listed: $N_1, \ldots, N_k$.
- Run the the 'parallel' automaton $N_1 \cup \cdots \cup N_k$ on some input string $x$.
- Choose the *smallest* $i$ such that we're in an accepting state of $N_i$. Choose class $L_i$ as the lexical class for $x$ with *highest priority*.
- Perform the specified *action* for the class $L_i$ (typically 'return tagged lexeme', or ignore).

Problem: How do we know when we've reached the end of the current lexical token?

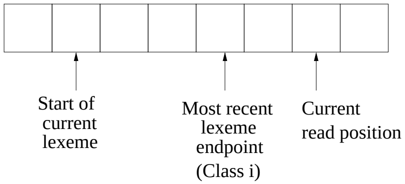
It needn't be at the *first* point where we enter an accepting state. E.g. `i`, `if`, `if2` and `if23` are all valid tokens in Java.

## Principle of longest match

In most computer languages, the convention is that each stage, the *longest possible* lexical token is selected. This is known as the principle of longest match (a.k.a. maximal munch).

To find the longest lexical token starting from a given point, we'd better run $N_1 \cup \cdots \cup N_k$ until it expires, i.e. the set of possible states becomes empty. (Or max lexeme length is exceeded. . . )

We'd better also keep a note of the *last* point at which we were in an accepting state (and what the top priority lexical class was). So we need to keep track of three positions in the text:



## Lexing: (conclusion)

Once our NFA has expired, we output the string from 'start' to 'most recent end' as a lexical token of class *i*.

We then advance the 'start' pointer to the character after the 'most recent end'. . . and repeat until the end of the file is reached.

All this is the basis for an efficient lexing procedure (further refinements are of course possible).

In the context of lexing, the same language definition will hopefully be applicable to hundreds of source files. So in contrast to pattern searching, well worth taking some time to 'optimize' our automaton.

## End-of-lecture challenge question

Very often in the process of lexing, as described on slide 17, it occurs that, at the time that the NFA expires, the most recent lexeme endpoint is only one character behind the current read position.

Question: Think of an example, taken from a real programming language, in which the most recent endpoint lies 2 or more characters behind the current read position.

## Reading

Relevant reading:

- Pattern matching: J & M chapter 2.1 is good. Also online documentation for `grep` and the like.
- Lexical analysis: see Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools*, Chapter 3.

Next time: Limitations of regular languages.