Search, Lexing and other applications

Informatics 2A: Lecture 7

Mary Cryan

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

1 October 2018

Where we're up to

In Lecture 6, we completed our study of regular expressions (and their equivalence to the languages recognized by Finite Automata) and started looking at some practical applications of regular language technology to string and pattern matching.

In this lecture, we will finish looking at pattern matching, and also discuss applications to data validation. Then we will look at:

- Lexical analysis of computer languages, etc. lexing. This is often the first stage of the language processing pipeline for computer languages (see Lecture 2).
- (Brief outline). Automatic verification of safety/liveness properties for (e.g. concurrent) finite-state systems.

grep, egrep, fgrep

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

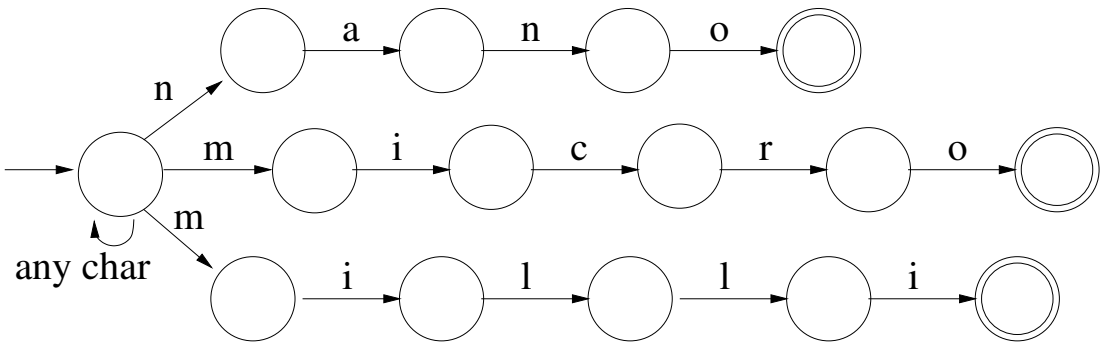
- `fgrep` searches for one or more fixed strings, using an efficient *string matching* algorithm.
- `grep` searches for strings matching a certain pattern (a simple kind of regular expression).
- `egrep` searches for strings matching an extended pattern (these give the full power of regular expressions).

On Friday, we saw how we can build a DFA to recognise a fixed string (ie, to implement `fgrep`).

Multiple (fixed) strings

Suppose now we want to find all occurrences of any of the strings nano, micro, milli in D.

No problem! Just do the same starting from the following NFA:



(The gain over the naive method is here readily apparent.)

To do more powerful searches, can use regular expressions . . .

Machine syntax for regular expressions

`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')

(The last three are allowed by `egrep`, but not by plain `grep`.)

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

Mathematical versus machine notation

We've now seen two notations for writing regular expressions:

- Mathematical notation, e.g. $(a + b)(a + b)^*$. This notation is intended to have as few operations as possible, for convenience in setting up the theory (e.g. Kleene algebra).
- Machine notation (regex), e.g. `(a|b)+`. This has a more generous set of operations, for convenience when writing complicated regular expressions.

The clashes between these are unfortunate, but we're stuck with them.

- Union of two languages is written using `|` in machine syntax, and $+$ in mathematical syntax.
- In machine syntax, $+$ is a unary operation representing concatenation of one or more strings of a given form.
- Dot `.` means concatenation in the mathematical syntax, and 'any character' in the machine syntax.

Example

How suitable are the patterns below for specifying the form of non-negative decimal integers?

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])*`

Example

How suitable are the patterns below for specifying the form of non-negative decimal integers?

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])*`

Answer: Pattern 1 is bad, because it admits the empty string.
Pattern 2 is fine if we don't mind leading zeros, e.g. 023.
Pattern 3 is just right for non-neg integers without leading zeros.
Pattern 4 is good for a common way of writing integers within English text, e.g. 1,024 or 578,000,000,000.

How egrep (typically) works

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

- Every machine regexp is clearly equivalent to a mathematical one.
- 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 at the end of Lecture 4.

  We don't 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 $+, ?, |$.

Regular expressions in data validation

Regexp's are used not just in searching, but also in checking whether data is of the expected form:

- Within XML documents, can enforce constraints on parts of the data:

```
<xs:simpleType name="ProductNumberType">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d{3}-[A-Z]{2}|\d7"/>
  </xs:restriction>
</xs:simpleType>
```

  (Example from P. Walmsley, *Definitive XML Schema*, 2012.)

- For text fields in web forms, check that the input text has the correct form. (See `regexlib.com` for hundreds of regexp's for validating email addresses, URLs, UK mobile phone numbers, postcodes, . . .)

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 does the (basic) regex language not include operations for intersection and complement?

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 does the (basic) regex language not include operations for intersection and complement?

Answer: If we included these, even a smallish regex could lead to a state space explosion. (Complement especially bad: need to determinize first!) The design of the regex language protects the unwary user from such nasty surprises.

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 does the (basic) regex language not include operations for intersection and complement?

Answer: If we included these, even a smallish regex could lead to a state space explosion. (Complement especially bad: need to determinize first!) The design of the regex language protects the unwary user from such nasty surprises.

WARNING: Some modern versions of so-called 'regex' (e.g. in Perl) include wild constructs that actually go way beyond the power of regular languages (back-references, backtracking, lookahead, recursive regexes) . . . and which definitely don't protect the unwary user from nastiness!

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`.

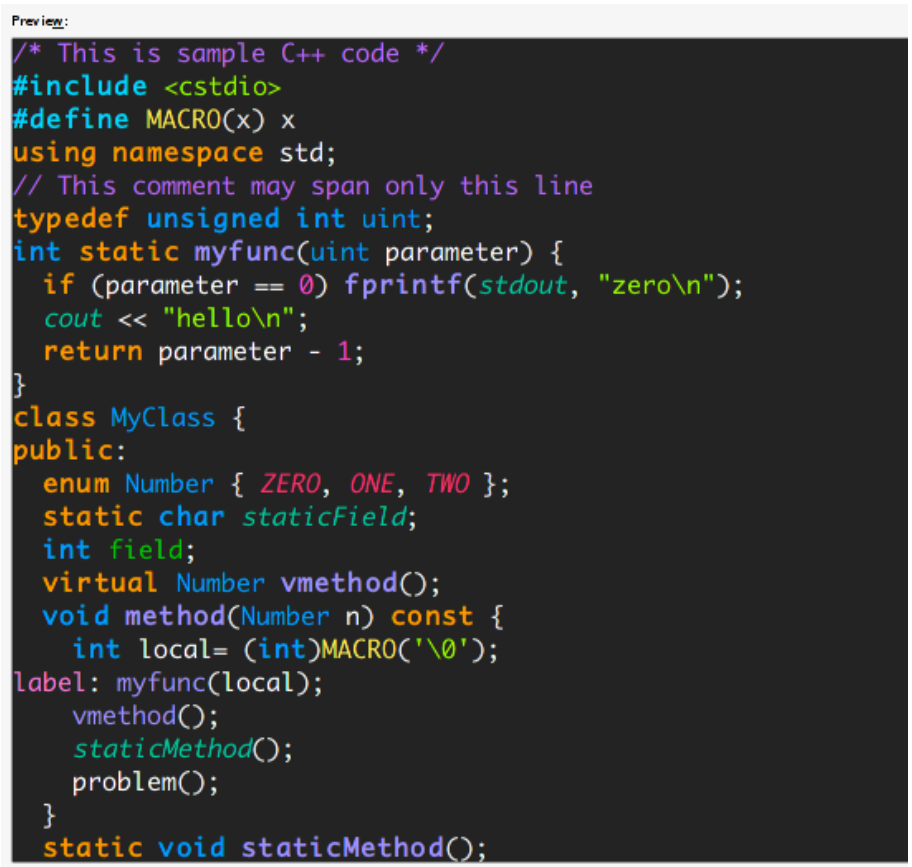
  Keywords of the language (like `while`) and other special symbols (like brackets) typically get a lexical class to themselves.

- Often, another job of the lexing phase is to throw away whitespace and comments. (E.g. in Java — but in Python, spacing matters!)

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

Syntax highlighting

Lexing doesn't just happen inside compilers and interpreters. Many modern editors/IDEs (e.g. Eclipse) do lexing as you type, for the purpose of syntax highlighting.



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}   { }
```

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 $N_1 \times \cdots \times N_k$ 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?

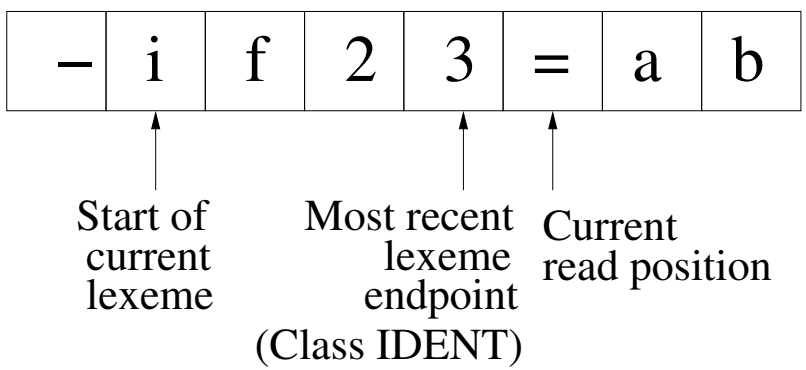
It needn't be at the *first* point where we enter an accepting state. E.g. `i`, `if`, `if2` and `if23` are all valid tokens in Java.

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 $N_1 \times \cdots \times N_k$ 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).

Hopefully the same lexer will be run on hundreds of source files. So probably worth taking the time to 'optimize' our automaton (e.g. by converting to a DFA, then minimizing.)

Finite automata and verification

Many concurrent systems arising in practice involve a bunch of finite-state processes that individually look quite simple.

But when put together, they can *interact* in very complex and subtle ways (large state space). Bugs can be hard to detect.

Regular language theory can help us to verify desirable properties automatically. E.g.

- Safety properties: "bad things don't happen"
- Liveness properties: "good things do happen"
- Fairness properties: "things good for some processes don't cause too much badness to others"

Simple example: Peterson's mutual exclusion protocol

Suppose we have two concurrent processes P_0, P_1 that may request access to some shared resource (e.g. a printer), but mustn't be given access at the same time.

P_0, P_1 can communicate using three shared flags:

- `req0` (initially `false`): 'whether P_0 wants access'.
- `req1` (initially `false`): 'whether P_1 wants access'.
- `turn` (values `0`,`1`): roughly, 'who is being allowed a turn'.

Code for P_0 when it wants access:

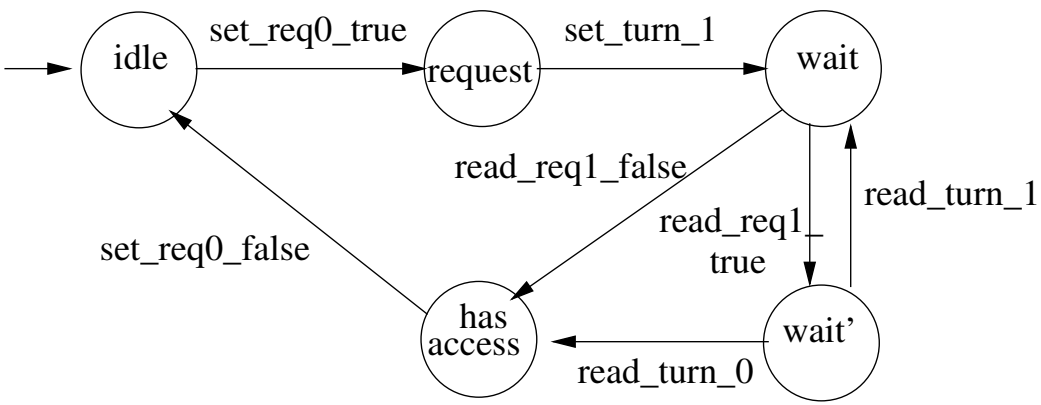
```
req0 = true ;
turn = 1 ;
while (req1 && turn == 1) { WAIT } ;
... // P0 now has access
req0 = false ;
```

Code for P_1 is same with $0, 1$ swapped and `req0,req1 swapped`.

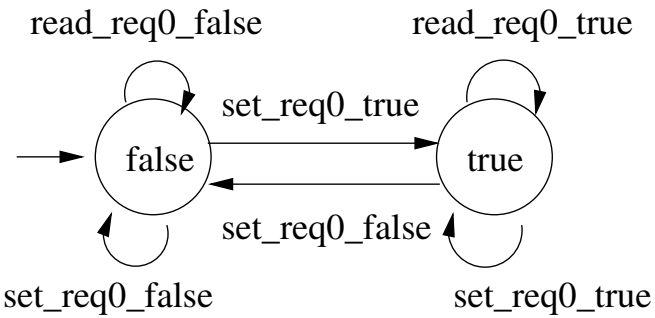
A finite-state model

We can model P_0, P_1 and each of the three flags by NFAs (constructed by hand). E.g.:

P0:



req0:



(All states are considered to be accepting.)

Combining the pieces

The 'language' for the complete system can now be obtained via a few standard constructions. (Here $\|$ denotes interleaving of regular languages—not officially defined in Inf2a.)

$$(\mathcal{L}(P_0) \parallel \mathcal{L}(P_1)) \cap (\mathcal{L}(\texttt{req0}) \parallel \mathcal{L}(\texttt{req1}) \parallel \mathcal{L}(\texttt{turn}))$$

The corresponding machine M can now be built automatically: 200 states in principle.

What's more, in a suitable logic, we can formulate properties like:

- Mutual exclusion (a safety property): P_0 and P_1 can never have access simultaneously.
- Progress (a liveness property): from any reachable state, *some* process can gain access if it tries.
- Bounded waiting (a fairness property): once P_0 has requested access, P_1 won't be given access *twice* before P_0 gets access.

There are algorithms for checking such properties automatically.

Next time ...

What sorts of things *can't* be done using regular languages?

How could we tell that some given language *isn't* regular?

We'll address these questions with a mathematical tool known as the Pumping Lemma — usually considered one of the hard bits in Inf2A ...