Lexing and other applications

Informatics 2A: Lecture 7

John Longley

School of Informatics
University of Edinburgh
`jrl@inf.ed.ac.uk`

4 October 2016

Where we're up to

In Lecture 6, we looked at some practical applications of regular language technology to string and pattern matching, and to data validation.

In this lecture, we'll look at some more:

- 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.

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.

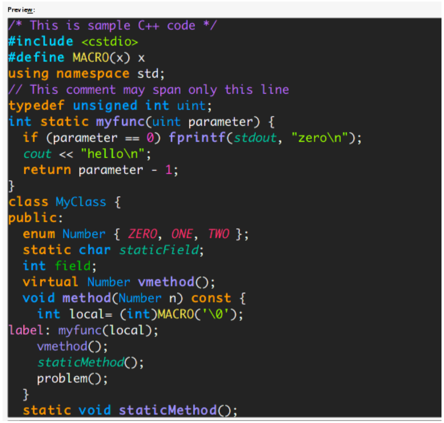
- Often, another job of the lexing phase is to throw away whitespace and comments. (E.g. in Java — but in Python, spacing matters!)

  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.



```
/* This is sample C++ code */
#include <cstdio>
#define MACRO(x) x
using namespace std;
// This comment may span only this line
typedef unsigned int uint;
int static myfunc(uint parameter) {
  if (parameter == 0) fprintf(stdout, "zero\n");
  cout << "hello\n";
  return parameter - 1;
}
class MyClass {
public:
  enum Number { ZERO, ONE, TWO };
  static char staticField;
  int field;
  virtual Number vmethod();
  void method(Number n) const {
    int local= (int)MACRO('\0');
label: myfunc(local);
    vmethod();
    staticMethod();
    problem();
  }
  static void staticMethod();
```

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]
```

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]
```

Answer: It matches certain other symbols such as '['. This is because the pattern `[A-z]` matches all characters from `A` to `z` inclusive within the ASCII character set — consult an ASCII table to see which characters these are.

Recognizing a lexical token using NFAs

- Build NFAs for our lexical classes $L_1, \dots, L_k$ in the order listed: $N_1, \dots, N_k$.
- 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?

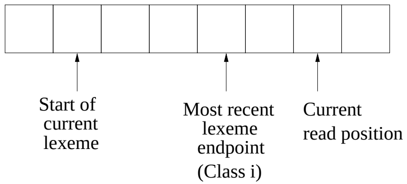
It needn't be at the *first* point where we enter an accepting state. E.g. `i`, `if`, `if2` and `if23` are all valid tokens in Java.

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).

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'.

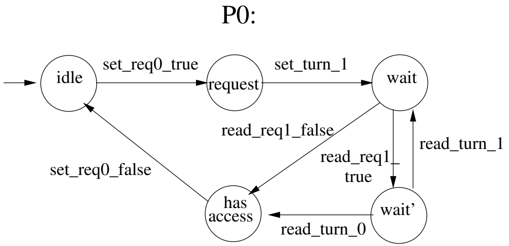
Code for P_0 when it wants access:

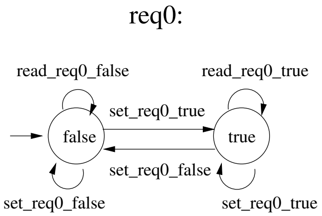
```
req0 = true ;
turn = 1 ;
while (req1 && turn == 1) { WAIT } ;
... // P0 now has access
req0 = false ;
```

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.:





(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.)

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

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 . . .