COMP261 Lecture 13

Lindsay Groves

Parsing 1 of 4:
Scanners and Regular Expressions

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Reading structured text

Lots of programs read complex data in text form. E.g.:

- *URLs*:
  http://ecs.vuw.ac.nz/lindsay/home.html
- E*mail addresses:*
  lindsay@ecs.vuw.ac.nz
- *Dates:*
  6/9/18, 6 September 2018, …
- *People's name, address, …*
- *Road information (as in Assignment 1 and 2)*
  node 10526 -36.87 174.69
  seg 17134 0.25 12420 12556 -36.88 174.72 -36.88

Reading structured text

- *SQL schema definition or query:*

```
DELETE FROM DomesticStudentsFor2017
    WHERE mark = 'E';
```

- *XML documents:*

```
<html><head><title>My Web Page</title></head>
<body><p>Thank you for viewing my page!</p>
</body></html>
```

- *Java statement:*

```
while ( A[k] != x ) { k++; }
```

Reading structured text

- Reading such data can be tedious and error prone

- Need systematic methods to help
  - Need to be able to describe the structure of the text
  - Use this description as basis for reading input, and for extracting its components

Describing structured text

- A *URL* is a string of the form http://name/name/…
  where each name is a string of letters, digits and dots.
- An *email address* is a string of the form id@id.id…
  where each id is a string of letters and digits.
- A *while statement* is a string of the form
  while (*exp*) *stmt*
  where *exp* is a valid expression and *stmt* is a valid statement
- A *statement* is either a *while statement*, an *if statement*, ….

Reading structured text

- We'll look at two ways of describing input structure, and ways of handling them:
  - Regular expressions and scanners
    Simple patterns with repetition and alternatives
    Supported directly by Java
  - Context free grammars and parsers
    More complex patterns with nesting
    Build parser based on the grammar

Using a Scanner for Lexical Analysis

- Java Scanner class allows you to read a string/file as a sequence of *tokens*.

```
while (scan.hasNext())
    System.out.println(scan.next());
```

- By default, a token is any sequence of characters, delimited by white space.

Using a Scanner for Lexical Analysis

- Can also read/test for more restricted kinds of tokens:

`hasNextInt()`	`nextInt()`
`hasNextShort()`	`nextShort()`
`hasNextLong()`	`nextLong()`
`hasNextDouble()`	`nextDouble()`
`hasNextFloat()`	`nextFloat()`
`hasNextBigInteger()`	`nextBigInteger()`
`hasNextBigDecimal()`	`nextBigDecimal()`
`hasNextByte()`	`nextByte()`
`hasNextBoolean()`	`nextBoolean()`
`hasNextLine()`	`nextLine()`

- `nextX` methods throw `InputMismatchException` if there isn't an X token in the input, so wrap reading in a `try` block.

Using a Scanner for Lexical Analysis

- What if we what to read other kinds of tokens?
- Or separate them in other ways?
- We can use patterns to describe the tokens we want to read:

```
while (scan.hasNext(pattern))
   System.out.println(scan.next(pattern));
```

- We can also specify a pattern to be used to delimit tokens:

```
scan.useDelimiter(pattern);
```

Specifying delimiters

- Delimiters are characters used to separate tokens, or to determine where tokens end.
- To separate tokens by single spaces:

```
    scan.useDelimiter(" ");
or: scan.useDelimiter("\\s");
```

- To separate tokens by multiple spaces:

```
    scan.useDelimiter("\\s+");
```

- To separate tokens by commas, or tabs:

```
    scan.useDelimiter(",");

    scan.useDelimiter("\\t");
```

Specifying delimiters

- Suppose we want to extract words from English text, ignoring spaces, punctuation, etc.

```
scan.useDelimiter("[\\w.,!()\"]+");
```

- `[abc]` describes a set of characters that can match
- `\s`, `\t`, `\w` match a space, tab, any white space
- `\` is used as “escape” character when following character has non-standard meaning.
- `x+` matches 1 or more occurrences of `x`.

Specifying delimiters

- If input file is:
  ```
  This text has:  some punctuation (some pointless!!); and some other "things".
  ```
- Tokens read are (with |'s around them):
  ```
  |This|, |text|, |has|, |some|, |punctuation|, |some|,|pointless|, |and|, |some|, |other|, |things|.
  ```
- How well does this work? Can we do better?
- Will it work for programming language text, eg:
  ```
  figure.walk(45,Math.min(Figure.stepSize,figure.curSpeed));
  ```
- What if we want to keep the brackets, comas, etc?

Using delimeters

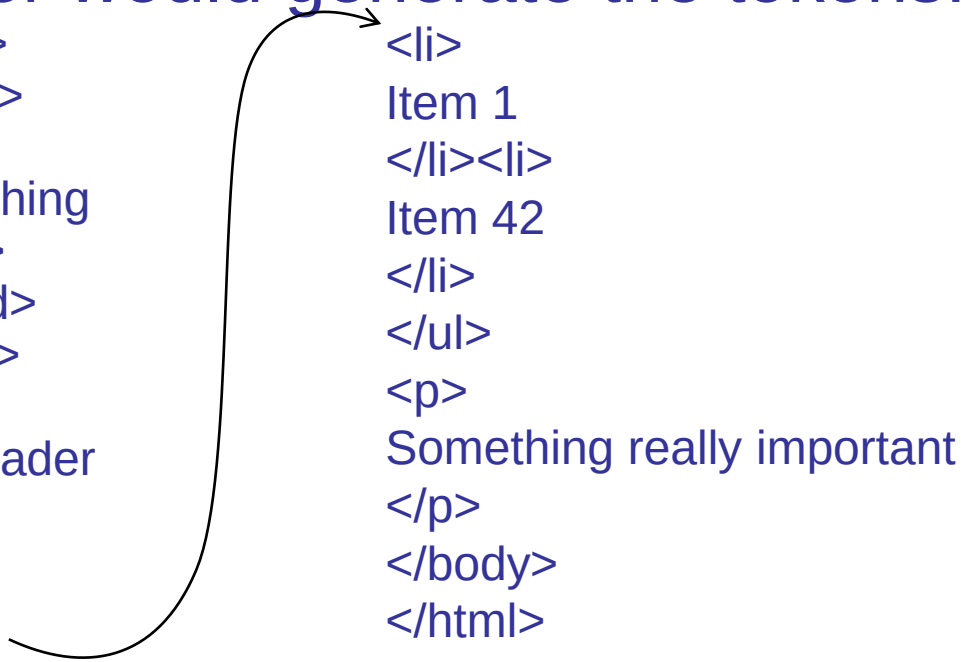
- Suppose we want to split an html file into tags and text between them.
- We can do this by defining and string of white space characters preceded by “>” or followed by “<“ as a delimiter.
- scan.useDelimiter("(?<=>)\\s*|\\s*(?=<)");
  - (?=end) (?<=begin) : pre- and post-context
  - | separates alternatives

Delimiter: "\\s*(?=<)|(?<=>)\\s*"

- Given:

```
<html>
<head><title> Something </title></head>
<body><h1> My Header </h1>
<ul><li> Item 1 </li><li> Item 42 </li></ul>
<p> Something really important </p>
</body>
</html>
```

- Scanner would generate the tokens:



Using delimeters: split

- Java strings have a split method, which breaks a string into tokens according to a pattern.

- ```
  String s = "lindsay@ecs.vuw.ac.nz";
  String w = s.split("[@.]");
  ```

- Ex: Implement split.
  ```

Lexical Analysis

- Defining delimiters can be very tricky.
  - Some languages (such as lisp, html, xml) are designed to be easy.

- Better approach:
  - Define a pattern to match the *tokens* (instead of a matching the *separators* between tokens)
  - Make a method that will search for and return the next token, based on the token pattern.
  - The pattern is typically made from combination of patterns for each kind of token – usually a regular expression.

- There are tools to make this easier:
  - eg LEX, JFLEX, ANTLR, …
  - see http://en.wikipedia.org/wiki/Lexical_analysis

Using a Scanner for Lexical Analysis

- Patterns are describe using ***Regular Expressions.***
- Simple patterns using sequence, alternatives (|) and repetition (*,+) – plus other extensions.

• abc	Matches “abc”
• a|e|i|o|u	One of a, e, i, o, u.
• .	Any character
• [0-9] [a-zA-Z] [+-*/]	Sets of possible characters
• \d \s	Digit, white space
• \.	Dot

For more details, see:
https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

Example Regular Expressions

- \d+ One or more digits
- \d+\.\d* One or more digits, followed by decimal point, followed by zero or more digits
- \d+(\.\d*)? One or more digits, optionally followed by decimal point, followed by zero or more digits
- [a-zA-z][a-zA-z0-9]* Letter followed by zero or more letters and/or digits
- \w+\s\w+ Two words separated by one space

Note: In Java, you need to use two \'s

How does it work?

- A regular expression is “compiled” into a kind a graph structure called a Finite Acceptor.
- Each edge is labelled with a character/symbol
- The acceptor can move from node A to node B if:
  - The next input character is, say x, and
  - There is an edge labelled x from A to B.
- Like a trie (lecture 3).

Compiling regular expressions

- Turning regular expressions into finite acceptors is expensive.
- Don’t want to do it every time you do a match.
- ```
  Pattern p = Pattern.compile("a*b");
  Matcher m = p.matcher("aaaaab");
  boolean b = m.matches();
  ```
- Can also call as:
  ```
  boolean b = Pattern.matches("a*b",
  "aaaaab");
  ```
  ```

For more information

- https://en.wikipedia.org/wiki/Regular_expression
- https://en.wikipedia.org/wiki/Deterministic_finite_automaton
- Lots of other tutorials, lecture notes, etc. on web