



COMP261 Parsing 2 of 4

Victoria  
UNIVERSITY OF WELLINGTON  
Te Whare Wānanga  
o te Ōpōtahi o te Ōhau o Aotearoa  
CAPITAL CITY UNIVERSITY

---

---

---

---

---

---

---

### How do we write programs to do this?

- The process of getting from the *input string* to the parse tree consists of *two steps*:
  - Lexical analysis*: the process of converting a sequence of characters into a sequence of tokens.
    - Note that `java.util.Scanner` allows us to do lexical analysis with great ease!
  - Syntactic analysis or parsing*: the process of analysing text, made of a sequence of tokens to determine its grammatical structure with respect to a given grammar.
    - Assignment will require you to write a recursive descent parser discussed in the next lecture!

---

---

---

---

---

---

---

### Using a Scanner for Lexical Analysis

- Need to separate the text into a sequence of tokens
- Java Scanner, by default, separates at white space.
 

```
figure.walk(45, Math.min(Figure.stepSize, figure.curSpeed));
```

 → white space is not good enough!!
- Java Scanner can use more complicated pattern to separate the tokens.
  - Can use a "Regular Expression"
    - string with "wild cards"
      - \* + ? : specifying repetitions
      - [~+\*] \d \s : specifying sets of possible characters
      - | : specifying alternatives
      - (?=end) (?<=begin) : specifying pre- and post-context
- eg: `scan.useDelimiter("(?<=>)\\s*\\s*(?=<=)");`

---

---

---

---

---

---

---

## Parsing text

Given

- some text,
- a grammar,
- a specification of the non-terminals of the grammar

First

- Lexical analysis: break up text into a sequence of tokens

Second

- Parsing:
  - (a) check if the text meets the grammar rules, or
  - (b) construct the parse tree for the text, according to the grammar.

## Lexical Analysis

The simplest approach: (spaces between tokens)

- Use the standard Java Scanner class
- Make sure that all the tokens are separated by white spaces (and don't contain any white spaces)
  - ⇒ the Scanner will return a sequence of the tokens
- very restricted: eg, couldn't separate tokens in html

More powerful approach:

- Use the standard Java Scanner class
- Define a delimiter that separates all the tokens
  - delimiter is a Java regular expression
  - text matching the delimiter will not be returned in tokens
  - eg
 

```
scan.useDelimiter("\\s*(?=<)|(?!>)\\s*");
```

 would separate the tokens for the html grammar:

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:

```
<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>
```

## 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 matching the *tokens* (instead of a pattern 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.
  - Patterns are generally regular expressions.
    - ⇒ use a finite state automata to match / recognise them.
- There are tools to make this easier:
  - eg LEX, JFLEX, ANTLR, ...
  - see [http://en.wikipedia.org/wiki/Lexical\\_analysis](http://en.wikipedia.org/wiki/Lexical_analysis)

---

---

---

---

---

---

---

---

## Parsing text?

- Consider this example grammar:
 

```
Expr ::= Num | Add | Sub | Mul | Div
Add  ::= "add" "(" Expr "," Expr ")"
Sub  ::= "sub" "(" Expr "," Expr ")"
Mul  ::= "mul" "(" Expr "," Expr ")"
Div  ::= "div" "(" Expr "," Expr ")"
Num  ::= an optional sign followed by a sequence of digits:
        [-+]?[0-9]+
```
- Check the following texts:
 

```
add(div( 56 , 8), mul(sub(0, 10 ), mul (-1, 3)))
div(div(86, 5), 67) 50
add(-5, sub(50, 50), 4)
div(100, 0)
```

---

---

---

---

---

---

---

---

## Idea: Write a Program to Mimic Rules!

- Naïve Top Down Recursive Descent Parsers:
    - have a method corresponding to each nonterminal that calls other nonterminal methods for each nonterminal and calls a scanner for each terminal!
- For example, given a grammar:
- ```
FOO ::= "a" BAR | "b" BAZ
BAR  ::= ....
```
- Parser would have a method such as:
- ```
public boolean parseFOO(Scanner s){
    if (!s.hasNext()) { return false; } // PARSE ERROR
    String token = s.next();
    if (token.equals("a")) { return parseBAR(s); }
    else if (token.equals("b")) { return parseBAZ(s); }
    else { return false; } // PARSE ERROR
}
```

---

---

---

---

---

---

---

---

