COMP261 Lecture 17

Parsing 2 of 4: Scanner and Parsing

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
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:*
  1. *Lexical analysis:* convert a sequence of characters into a sequence of tokens.
     - Note that java.util.Scanner allows us to do lexical analysis with great ease!
  2. *Syntactic analysis or parsing:* analyse 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!

Parsing text

Given: a grammar, some text to be parsed:

First: Lexical analysis / Scanning / Tokenising

- Break up text into a sequence of tokens
- Remove white space

Second: Syntax analysis / Parsing

- Check if the text meets the grammar rules, OR
- Construct the parse tree for the text, according to the grammar.

Using a Scanner for Lexical Analysis

- Need to separate the text into a sequence of tokens
- Java Scanner class acts as a scanner, breaking a string/file into a sequence of tokens.

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

- By default, separates at white space, which is ok for many text applications, but not for programming languages, eg:

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

Using a Scanner for Lexical Analysis

- Java Scanner can use a ***Regular Expression*** to separate the tokens.
  - string with “wild cards”
  - [-+*/] [0-9] \d \s : sets of possible characters
  - | : alternatives
  - * + ? : repetition (>=0, >=1, 0 or 1)
  - (?=end) (?<=begin) : pre- and post-context
- E.g.: scan.useDelimiter("(?<=>)\\s*|\\s*(?=<)");
  - Treats and string of white space characters preceded by “>” or followed by “<“ as a delimiter.
  - Can use for html file

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:

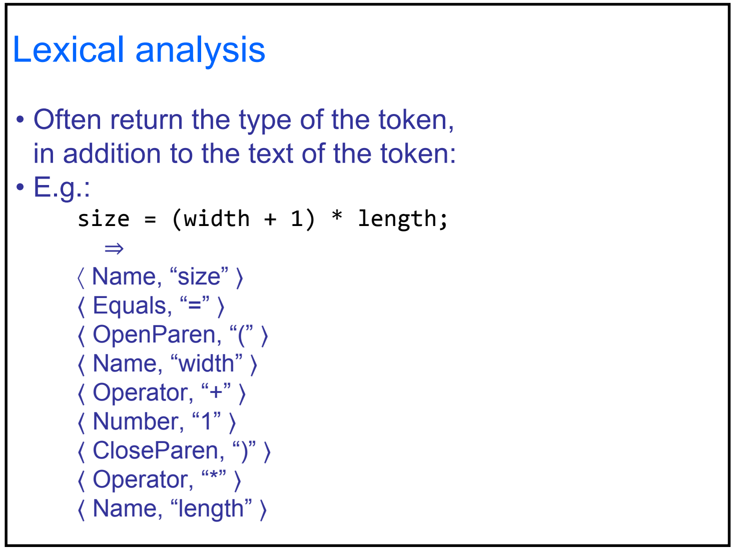
```
<html>
<head><title> Something </title></head>
<body><h1> My Header </h1>
<ul><li> Item 1 </li><li> Item 42 </li></ul>
<p> Something really important </p>
</body>
</html>
```

- 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 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.
    ⇒ use a finite state automaton to match / recognise them.
- There are tools to make this easier:
  - eg LEX, JFLEX, ANTLR, …
  - see http://en.wikipedia.org/wiki/Lexical_analysis



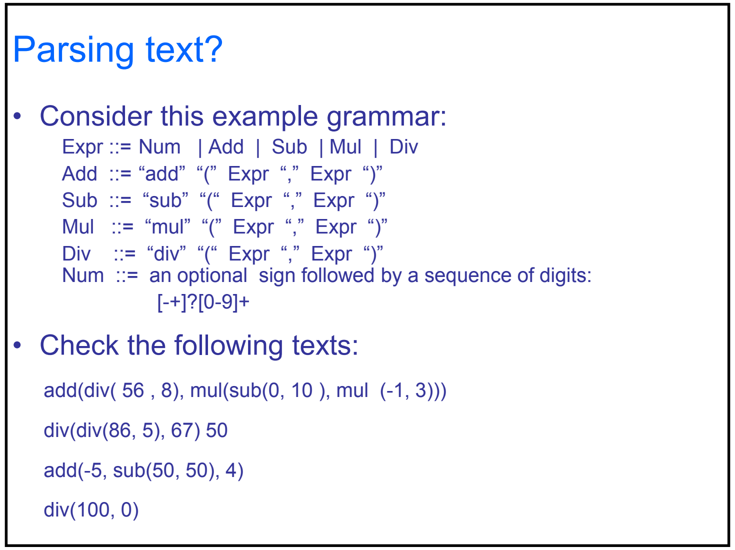
Lexical analysis

- Often return the type of the token, in addition to the text of the token:
- E.g.:

```
size = (width + 1) * length;
```

⇒

⟨ Name, “size” ⟩
⟨ Equals, “=” ⟩
⟨ OpenParen, “(” ⟩
⟨ Name, “width” ⟩
⟨ Operator, “+” ⟩
⟨ Number, “1” ⟩
⟨ CloseParen, “)” ⟩
⟨ Operator, “*” ⟩
⟨ Name, “length” ⟩



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]+
```

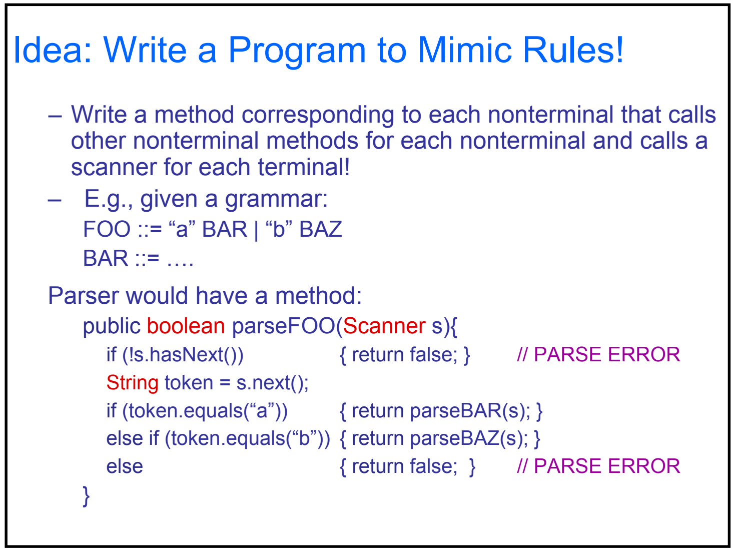
- Check the following texts:

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)

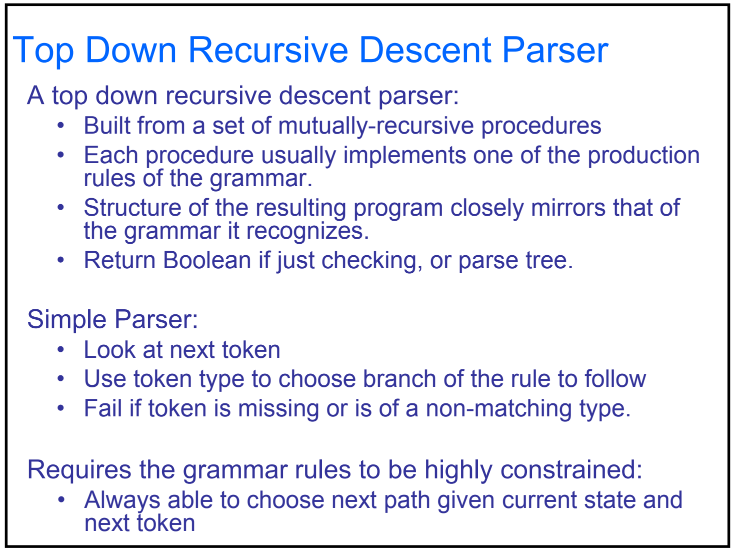


Idea: Write a Program to Mimic Rules!

- Write a method corresponding to each nonterminal that calls other nonterminal methods for each nonterminal and calls a scanner for each terminal!
- E.g., given a grammar:
  FOO ::= “a” BAR | “b” BAZ
  BAR ::= ….

Parser would have a method:

```
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
}
```



Top Down Recursive Descent Parser

A top down recursive descent parser:

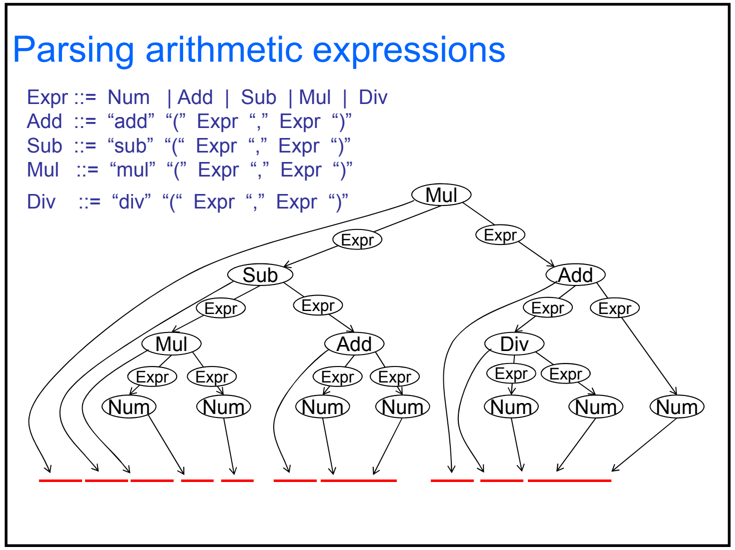
- Built from a set of mutually-recursive procedures
- Each procedure usually implements one of the production rules of the grammar.
- Structure of the resulting program closely mirrors that of the grammar it recognizes.
- Return Boolean if just checking, or parse tree.

Simple Parser:

- Look at next token
- Use token type to choose branch of the rule to follow
- Fail if token is missing or is of a non-matching type.

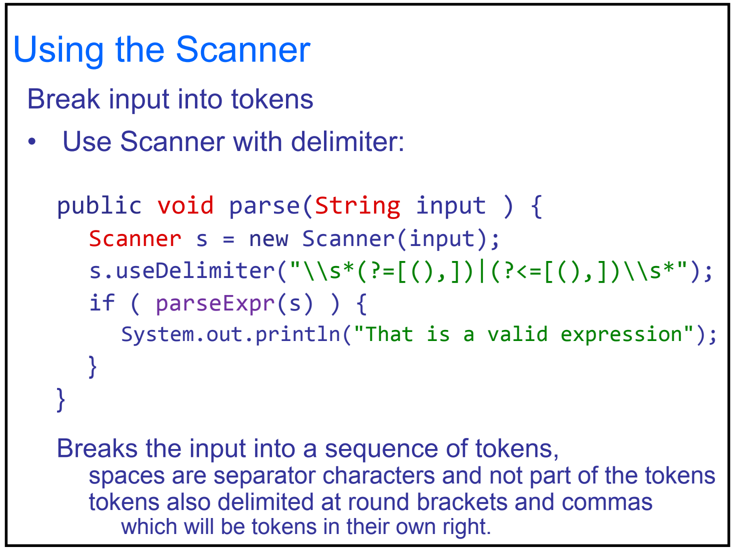
Requires the grammar rules to be highly constrained:

- Always able to choose next path given current state and next token



Parsing arithmetic expressions

Expr ::= Num | Add | Sub | Mul | Div
Add ::= “add” “(” Expr “,” Expr “)”
Sub ::= “sub” “(“ Expr “,” Expr “)”
Mul ::= “mul” “(” Expr “,” Expr “)”
Div ::= “div” “(“ Expr “,” Expr “)”



Using the Scanner

Break input into tokens

- Use Scanner with delimiter:

```
public void parse(String input ) {
   Scanner s = new Scanner(input);
   s.useDelimiter("\\s*(?=[(),])|(?<=[(),])\\s*");
   if ( parseExpr(s) ) {
      System.out.println("That is a valid expression");
   }
}
```

Breaks the input into a sequence of tokens,
spaces are separator characters and not part of the tokens
tokens also delimited at round brackets and commas
which will be tokens in their own right.