COMP261 Parsing 2 of 4	
81	
3[
Victoria	
To Where Williams or to Fifth and Administration of the Land Administration	
	1
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.cur Speed));	
→ 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	
[-+*/] \(\) \(\) is specifying sets of possible characters : specifying alternatives	
(?=end) (?<=begin) : specifying pre- and post-contexteg: 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.	
	7
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 htmlMore powerful approach:	
Use the standard Java Scanner classDefine 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:	
<pre><head><title> Something </title></head> <body><h1> My Header </h1> ltem 1 /li>ltem 42 /p> Something really important </body></pre>	
scanner would generate the tokens: 	
<head></head>	
<pre> </pre> <pre></pre> <pre></pre> <pre></pre> <pre>My Header</pre> Something really important	
My Header Something really important Abody Abody Abody 	

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 Parsing text? Consider this example grammar: Consider this example grantman. 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:

// PARSE ERROR

// PARSE ERROR

FOO ::= "a" BAR | "b" BAZ

if (!s.hasNext())

String token = s.next(); if (token.equals("a"))

Parser would have a method such as: public boolean parseFOO(Scanner s){

{ return false; }

{ return false; }

else if (token.equals("b")) { return parseBAZ(s); }

{ return parseBAR(s); }

BAR ::=

else

Top Down Recursive Descent Parser

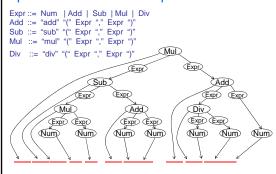
A top down recursive descent parser:

- 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.

Naive Parser:

- looks at next token
- checks what the token is to decide which branch of the rule to follow
- fails 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

A parser for arithmetic expressions



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");
    }
}</pre>
```

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.