Class Rep Election

- Alan Elliot John Ke Ryan Zaki

Admin

- Nuku quiz
- Tutorials and help desks start next week

How do we write a parser program to do this?

- The process of getting from the *input string* to the parse tree consists of *two steps:*
  1. Lexical analysis:
     The process of converting a sequence of characters into a sequence of tokens.
     - Note that java.util.Scanner allows us to do simple lexical analysis quite easily!
  2. Syntactic analysis or parsing:
     The process of analysing a sequence of tokens to determine its grammatical structure with respect to a given grammar.

Lexical Analysis: Using a Scanner

- 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" (java.util.regexp.Pattern)
    - string with "wild cards"
    - [-+*/]   \d \s : specifying sets of possible characters
    - | : specifying alternatives
    - * + ? : specifying repetitions
    - (?<=before) (?=after) : specifying pre-context and post-context
- eg: scan.useDelimiter("\\s*(?=[<])|(?<=[>])\\s*")
  scan.useDelimiter("\\s+|(?=[{}(),;])|(?<=[{}(),;])")

A Scanner Delimiter: "\\s*(?=[<])|(?<=[>])\\s*"

- Given:

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

- Scanner with "\\s*(?=[<])|(?<=[>])\\s*" delimiter would generate the tokens:

  <html>
  <head>
  <title>
  Something
  </title>
  </head>
  <body>
  <h1>
  My Header
  </h1>

  Item 1

  Item 42

  <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.

- Alternative 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.
    - ⇒ compiled into 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

- Analysing a sequence of tokens with respect to a given grammar.

- Levels of parsing:
  - Does the text conform to the grammar? (Yes/No)
  - Construct an Abstract Syntax Tree (or fail)
  - Construct an Abstract Syntax Tree or report the errors where the text is ungrammatical.

- There are lots of different parsing algorithms:
  - Top-down vs bottom-up
  - List from Wikipedia: Canonical LR, Chart, CYK algorithm, Earley, GLR, Inside–outside algorithm, LALR, Left corner, LL, LR, Operator-precedence, Packrat, PQCC, Recursive ascent, **Recursive descent**, Scannerless, Shift-reduce, Shunting yard, Simple precedence,Tail recursive parser
  - Assignment will require you to write a recursive descent parser

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]+"
- Example texts: (are they valid?)
  - add(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)

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: (unambiguous, "LL(1)")
  - always able to choose next path given current state and next token

The Program Mimics the Grammar 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:

SENT ::= “the” DEFNP | “a” INDEFNP

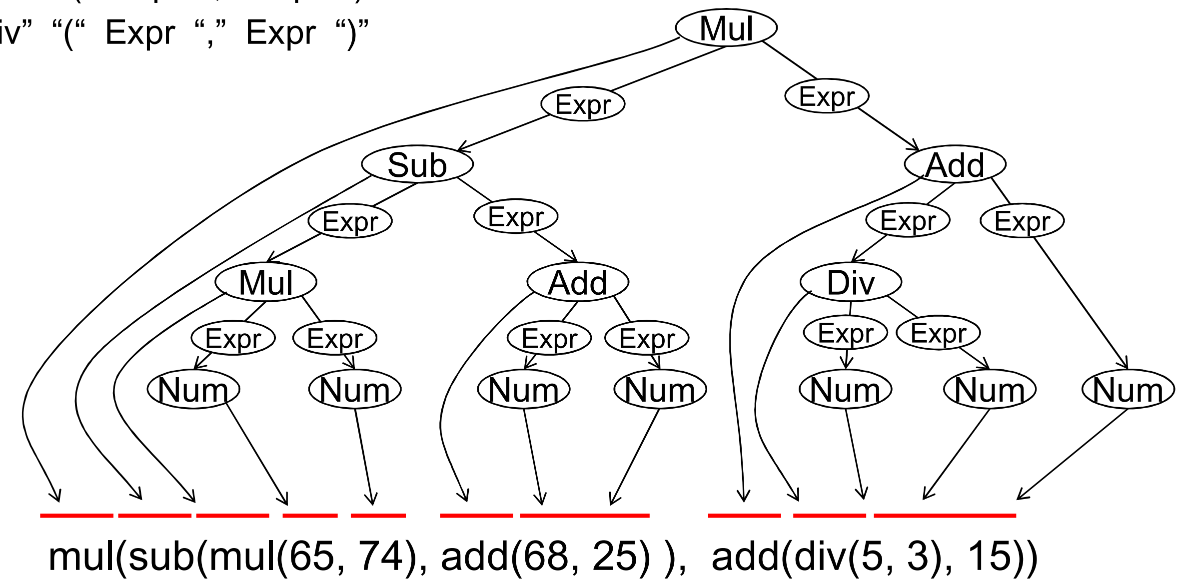
DEFNP ::= ….

Parser (just to check a text) would have a method such as:

```
public boolean parseSENT(Scanner s){
    if (s.hasNext(“the”))           { s.next();  return parseDEFNP(s); }
    else if (s.hasNext(“a”))        { s.next();  return parseINDEFNP(s); }
    else                            { return false;  }
}
```

A parser for 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+|(?=[(),])|(?<=[(),])");
   if ( parseExpr(s) ) {System.out.println("That is a valid expression");}
   else                {System.out.println("That expression is NOT valid");}
}
```

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.

Looking at next token

- Need to be able to look at the next token to work out which branch to take:
  - Scanner has two forms of hasNext:
    - s.hasNext():
      → is there another token in the scanner?
    - s.hasNext("*string to match*"):
      → is there another token, and does it match the string?

```
if ( s.hasNext("add") ) { …..
```

  - Can use this to peek at the next token without reading it
  - String can be a regular expression!

```
if ( s.hasNext("[-+]?[0-9]+") ) { …..
```

    - true if the next token is an integer
  - Good design for parser because the next token might be needed by another rule/method if it isn't the right one for this rule/method.

Parsing Expressions (checking only)

Expr ::= Num | Add | Sub | Mul | Div

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext("[-+]?[0-9]+")) { s.next(); return true; }     // Num
   if (s.hasNext("add"))         { return parseAdd(s); }
   if (s.hasNext("sub"))         { return parseSub(s); }
   if (s.hasNext("mul"))         { return parseMul(s); }
   if (s.hasNext("div"))         { return parseDiv(s); }
   return false;
 }
```

Parsing Expressions (checking only)

Add ::= “add” “(” Expr “,” Expr “)”

```
public boolean parseAdd(Scanner s) {
   if (s.hasNext(“add”)) {s.next();}    else {return false; }
   if (s.hasNext(“(”))   {s.next();}    else {return false; }
   if (parseExpr(s))     { }            else {return false; }
   if (s.hasNext(“,”))   {s.next();}    else {return false; }
   if (parseExpr(s))     { }            else {return false; }
   if (s.hasNext(“)”))   {s.next();}    else {return false; }
   return true;
}
```

Parsing Expressions (checking only)

Sub ::= “sub” “(“ Expr “,” Expr “)”

```
public boolean parseSub(Scanner s) {
   if (s.hasNext(“sub”)) {s.next();} else {return false;}
   if (s.hasNext(“(”))   {s.next();} else {return false;}
   if (parseExpr(s))     { }         else {return false;}
   if (s.hasNext(“,”))   {s.next();} else {return false;}
   if (parseExpr(s))     { }         else {return false;}
   if (s.hasNext(“)”))   {s.next();} else {return false;}
   return true;
}
```

same for parseMul and parseDiv

Cleaning up the code (checking only)

```
public boolean parseAdd(Scanner s) {
   if (s.hasNext("add")) {s.next();}    else {return false; }
   if (s.hasNext("("))   {s.next();}    else {return false; }
   if (parseExpr(s))     { }            else {return false; }
   if (s.hasNext(","))   {s.next();}    else {return false; }
   if (parseExpr(s))     { }            else {return false; }
   if (s.hasNext(")"))   {s.next();}    else {return false; }
   return true;
}
```

// consumes next token if it matches pat, reports error if not

```
public boolean checkFor(String pat, Scanner s){
    if (s.hasNext(pat)) {s.next(); return true}
    else {return false;}
}
```

Cleaning up the code (checking only)

```
public boolean parseAdd(Scanner s) {
   if (!checkFor("add", s))   {return false; }
   if (!checkFor("(", s))     {return false; }
   if (!parseExpr(s))         {return false; }
   if (!checkFor(",", s))     {return false; }
   if (!parseExpr(s))         {return false; }
   if (!checkFor(")", s))     {return false; }
   return true;
}

// consumes next token if it matches pat, doesn't if not matching
public boolean checkFor(String pat, Scanner s){
    if (s.hasNext(pat)) {s.next(); return true}
    else {return false;}
}
```

Better coding: using patterns

- Give names to patterns to make program easier to understand and to modify
- Precompile the patterns for efficiency:

```
private static final Pattern NUM_PAT = Pattern.compile("[-+]?[0-9]+");
private static final Pattern ADD_PAT = Pattern.compile("add");
private static final Pattern SUB_PAT = Pattern.compile("sub");
private static final Pattern MUL_PAT = Pattern.compile("mul");
private static final Pattern DIV_PAT = Pattern.compile("div");
private static final Pattern OPEN_PAT = Pattern.compile("\\(");
private static final Pattern COMMA_PAT = Pattern.compile(",");
private static final Pattern CLOSE_PAT = Pattern.compile("\\)");
```

Using patterns (checking only)

```
public boolean parseAdd(Scanner s) {
   if (!checkFor(ADD_PAT, s))     {return false; }
   if (!checkFor(OPEN_PAT, s))    {return false; }
   if (!parseExpr(s))             {return false; }
   if (!checkFor(COMMA_PAT, s))   {return false; }
   if (!parseExpr(s))             {return false; }
   if (!checkFor(CLOSE_PAT, s))   {return false; }
   return true;
}
```

```
// consumes next token if it matches pat, doesn't if not matching
public boolean checkFor(Pattern pat, Scanner s){
    if (s.hasNext(pat)) {s.next(); return true}
    else {return false;}
}
```

Using Patterns (checking only)

Expr ::= Num | Add | Sub | Mul | Div

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext(NUM_PAT))       { s.next(); return true; }     // Num
   if (s.hasNext(ADD_PAT))       { return parseAdd(s); }
   if (s.hasNext(SUB_PAT))       { return parseSub(s); }
   if (s.hasNext(MUL_PAT))       { return parseMul(s); }
   if (s.hasNext(DIV_PAT))       { return parseDiv(s); }
   return false;
 }
```