COMP261 Lecture 14

Lindsay Groves

Parsing 2 of 4: Grammars and Parsing

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

CAPITAL CITY UNIVERSITY

Reading structured text – take two

- Now consider more complex forms of text:
  - *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++; }
    ```
- More complex structures get cumbersome to describe with regular expressions.
- Some patterns, such as nested structures, can’t be expressed with regular expressions.

Describing structured text: Grammars

- A **grammar** is a set of rules, describing the structure of strings in a formal language (set of strings).
- The rules describe how to form strings in the language, and names its structural components.
  whileStmt ::= “while” “(“ condition “)” statement
- Can show alternative forms, and can be recursive:
  statement ::= whileStmt | ifStmt | …
- A grammar only describes the form of allowable strings, not their meaning.
- Official name is **context-free grammar** – look it up!

Example: Java statements (simplified!!)

Statement ::=

Variable “=” *Exp* “;” |

“if” “(“ *Exp* “)” *Statement* |

“if” “(“ *Exp* “)” *Statement* “else” *Statement* |

“while” “(” *Exp* “)” *Statement* |

“do” *Statement* “while” “(“ *Exp* “)” |

“{“ *Statement-list* “}” |

…

Exp ::= *Variable* | *Constant* | …

Ex: Look on-line for full Java grammar.

Example: A simple html grammar

HTMLFILE ::= “<html>” [HEAD] BODY “</html>”

HEAD ::= “<head>” TITLE “</head>”

TITLE ::= “<title>” TEXT “</title>”

BODY ::= “<body>” [BODYTAG]* “</body>”

BODYTAG ::= H1TAG | PTAG | OLTAG | ULTAG

H1TAG ::= “<h1>” TEXT “</h1>”

PTAG ::= “<p>” TEXT “</p>”

OLTAG ::= “<ol>” [LITAG]+ “</ol>”

ULTAG ::= “<ul>” [LITAG]+ “</ul>”

LITAG ::= “<li>” TEXT “</li>”

TEXT ::= [.[^<>]]+ (*Sequence of characters other than < and >*)

Grammar structure: Terminals

- Literal strings or patterns of characters that can occur in texts
- Here they are enclosed in double quote marks
- Classes of terminals (like numbers, identifiers) defined by RE’s.

```
HTMLFILE ::= “<html>”  [ HEAD ]  BODY “</html>”
HEAD ::= “<head>”  TITLE  “</head>”
TITLE ::= “<title>”  TEXT  “</title>”
BODY ::= “<body>”  [ BODYTAG ]*  “</body>”
BODYTAG ::= H1TAG  |  PTAG  |  OLTAG  |  ULTAG
H1TAG ::= “<h1>” TEXT  “</h1>”
PTAG ::= “<p>”  TEXT  “</p>”
OLTAG ::= “<ol>”  [ LITAG ]+  “</ol>”
ULTAG ::= “<ul>”  [ LITAG ]+  “</ul>”
LITAG ::= “<li>”  TEXT  “</li>”
TEXT ::=  [.[^<>]]+   (Sequence of  characters other than < and >)
```

Grammar structure: Nonterminals

- Name structural components of strings (not part of the text)
- Defined by rules.

Top level nonterminal (start symbol) usually first

```
HTMLFILE ::= "<html>"  [ HEAD ]  BODY "</html>"
HEAD ::= "<head>"  TITLE  "</head>"
TITLE ::= "<title>"  TEXT   "</title>"
BODY ::= "<body>"  [ BODYTAG ]*  "</body>"
BODYTAG ::= H1TAG | PTAG  | OLTAG | ULTAG
H1TAG ::= "<h1>" TEXT  "</h1>"
PTAG ::= "<p>"  TEXT  "</p>"
OLTAG ::= "<ol>"  [ LITAG ]+  "</ol>"
ULTAG ::= "<ul>"  [ LITAG ]+  "</ul>"
LITAG ::= "<li>"  TEXT  "</li>"
TEXT ::= [.[^<>]]+
```

(Sequence of characters other than < and >)

Grammar structure: Meta-symbols

- Meta-symbols are fixed parts of the grammar notation.
- ::= = is defined as, **|** = “or”
- […] = optional, […]* = zero or more, **[…]+** = one or more

```
HTMLFILE ::= “<html>”  [ HEAD ]  BODY “</html>”
HEAD ::= “<head>”  TITLE  “</head>”
TITLE ::= “<title>”  TEXT  “</title>”
BODY ::= “<body>”  [ BODYTAG ]*  “</body>”
BODYTAG ::= H1TAG  |  PTAG  |  OLTAG  |  ULTAG
H1TAG ::= “<h1>” TEXT  “</h1>”
PTAG ::= “<p>”  TEXT  “</p>”
OLTAG ::= “<ol>”  [ LITAG ]+  “</ol>”
ULTAG ::= “<ul>”  [ LITAG ]+  “</ul>”
LITAG ::= “<li>”  TEXT  “</li>”
TEXT ::= [.[^<>]]+ (Sequence of  characters other than < and >)
```

Using the Grammar

Given some text:

```
<html>
<head><title> Today</title></head>
<body><h1> My Day </h1>
<ul><li>meeting</li><li> lecture </li></ul>
<p> parsing stuff</p>
</body>
</html>
```

- Is it a valid piece of HTML?
  - Does it conform to the grammar rules?
- What is the structure? (Needed in order to process it)
  - what are the components?
  - what types are the components?
  - how are they related?

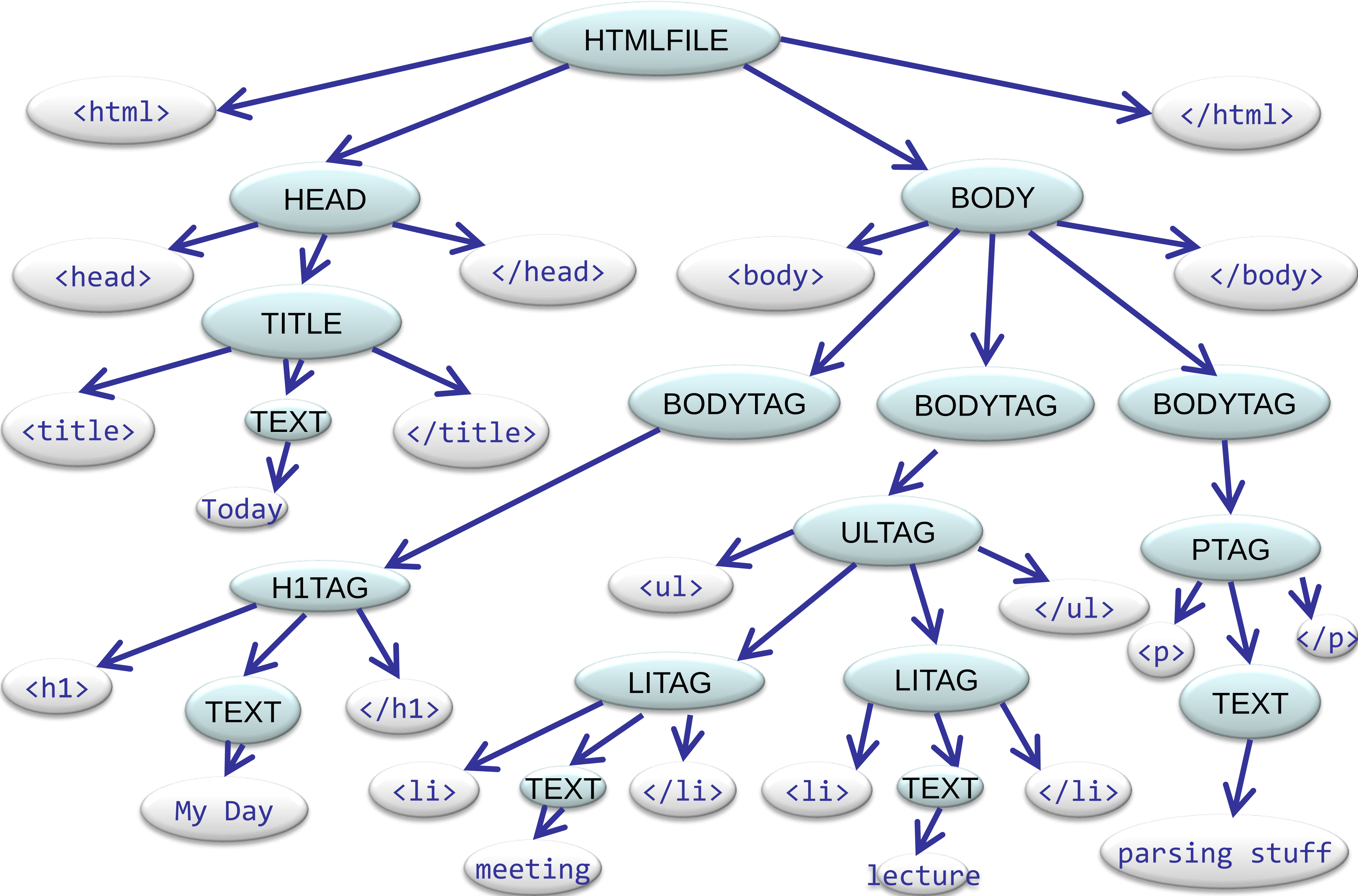
What kind of structure?

- Structure of a text is hierarchical

- Can be described by an **ordered tree**
  - Leaves correspond to terminals.
  - Internal nodes labelled with nonterminals.
  - Root is labelled with the start symbol.
  - Each internal node and its children correspond to a grammar rule (or an alternative in a grammar rule).
  - The text consists of all terminals on the **fringe** of the tree.

- A (**concrete) syntax tree** or **parse tree** represents the syntactic structure of a string according to the grammar, showing all the components of the rules.

Parse Tree


HTMLFILE
<html>
</html>
HEAD
BODY
<head>
</head>
TITLE
<body>
</body>
<title>
TEXT
</title>
Today
BODYTAG
BODYTAG
BODYTAG
ULTAG
PTAG
H1TAG
<ul>
</ul>
<p>
</p>
<h1>
TEXT
</h1>
LITAG
LITAG
TEXT
My Day
<li>
TEXT
</li>
<li>
TEXT
</li>
meeting
lecture
parsing stuff

Grammars define possible parse trees

- Each grammar rule defines possible structures that may occur in a parse tree.

```
H1TAG ::= “<h1>”  TEXT  “</h1>”

BODYTAG ::= H1TAG  |  PTAG  |  OLTAG  |  ULTAG

HTMLFILE ::= “<html>”  [ HEAD ]  BODY “</html>”
```

- A text is in the language defined by a grammar iff you can construct a parse tree for it.

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  ::=  [-+]?[0-9]+
```

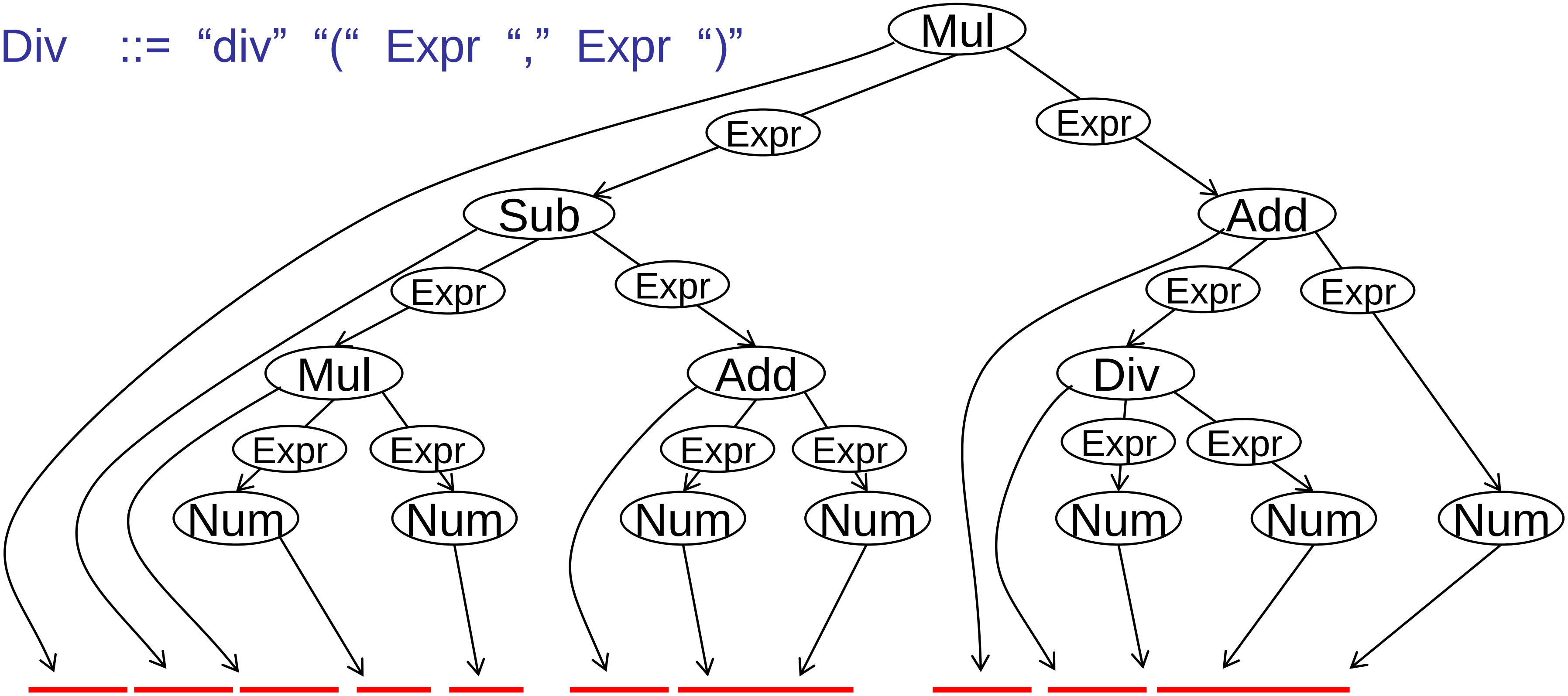
(an optional sign followed by a sequence of digits)

- 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)
```

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 “)”



How do we write programs to do this?

Given: a grammar, and some text to be parsed:

1: Lexical analysis / Scanning / Tokenising

- Break up text into a sequence of tokens
- Remove white space (and comments)

2: Syntax analysis / Parsing

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

- Separating scanning simplifies the parser.

Using a Scanner for Lexical Analysis

- We can use a scanner, as describe in lecture 13.
- Can read tokens one at a time as they are required by the parser,
- Or read the whole file/text and turn it into a list of tokens before parsing starts.

Idea: Write a Program to Mimic Rules!

- Write a parse method corresponding to each nonterminal that calls other methods for each nonterminal and calls the 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 rule of the grammar.
- Structure of the resulting program closely mirrors that of the grammar.
- 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.

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.

Example: Simple expressions

- Consider the following 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]+
```

- What does a parser based on this grammar look like?
  - – There is a method for each non terminal.
  - – They need to follow the structure of the grammar rules.

Parser for expressions - first attempt

```
public boolean parseExpr(Scanner s) {
  if ( !s.hasNext() )  { return false; }          // PARSE ERROR
  String token = s.next();
  if ( token is a number ) { return true; }
  if ( token = “add” ) { return parseAdd(s); }
  if ( token = “sub” ) { return parseSub(s); }
  if ( token = “mul” ) { return parseMul(s); }
  if ( token = “div” )  { return parseDiv(s); }
  else                  { return false;  }        // PARSE ERROR
}


public boolean parseAdd(Scanner s) {
  if ( !s.hasNext() )   { return false; }         // PARSE ERROR
  String token = s.next();
  if ( token != “add” ) {return false; }          // PARSE ERROR
  token = s.next();
  if ( token != “(“ )     {return false; }        // PARSE ERROR
  …
```

What’s wrong here??

Accessing the next token

- How does parseAdd access the next token, when parseExpr has already read it?

- If you read the next token to test it, it’s no longer there for another method to inspect!

- One approach: Could implement an alternative scanner class with *current* and *advance* methods.

Accessing the next token

- A second approach: Save the next token ("lookahead") in a field of a parser object, which contains the parser methods.
- Can keep the scanner in a field too, rather than pass it to every parser method.
-
  ```
  public class Parser {
    Scanner s;
    Token t = null;
    public Parser(Scanner scanner) { s = scanner; }
    public parseExp() { … }
    …
  }
  ```

Looking at next token

- A third approach: Check for specific kinds of tokens. So lookahead token remains in the inout.

- Scanner can test for a particular kind of token: hasNextInt, hasNextFloat, hasNextBoolean, …

- Can also check for a particular string:
  - s.hasNext("string to match"):

    → is there another token, and does it match the string?

    ```
    if ( s.hasNext("add") ) { …..
    ```

- Or for a regular expression:

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

  - true if there is another token, which is an integer

Parsing Expressions (checking only)

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext(“[-+]?[0-9]+”))     { s.next(); return true; }
   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;
}
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))                          { return false; }
   if (s.hasNext(“,”))      { s.next(); } else { return false; }
   if (!parseExpr(s))                          { return false; }
   if (s.hasNext(“)”))      { s.next(); } else { return false; }
   return true;
}
```

Parsing Expressions (checking only)

```
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))                        { return false; }
   if (s.hasNext(“,”))    { s.next(); } else { return false; }
   if (!parseExpr(s))                        { return false; }
   if (s.hasNext(“)”))    { s.next(); } else { return false; }
   return true;
}
```

same for parseMul and parseDiv

Parsing Expressions (checking only)

Alternative, given similarity of Add, Sub, Mul, Div:

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext(“[-+]?[0-9]+”))    { s.next();  return true; }
   if (s.hasNext(“add|sub|mul|div”)) {s.next();} else {return false;}
   if (s.hasNext(“(”))  { s.next(); } else  { return false; }
   if (!parseExpr(s))                       { return false; }
   if (s.hasNext(“,”))  { s.next(); } else  { return false; }
   if (!parseExpr(s))                       { return false; }
   if (s.hasNext(“)”))  { s.next(); } else  { return false; }
   return true;
}
```

This amounts to changing the grammar to:

Expr ::= Num | Op “(” Expr “,” Expr “)”
Op ::= “add” | “sub” | “mul” | “div”
Num ::= [-+]?[0-9]+

And writing the code for parseOP and parseNum inline.

Simplifying the parser

We can reduce the duplication in checking for terminals:

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext(“[-+]?[0-9]+”)) { s.next(); return true; }
   require(s, “add|sub|mul|div”));
   require(s, “(”);
   if (!parseExpr(s)) { return false; }
   require(“,”);
   if (!parseExpr(s)) { return false; }
   require(s, “)”);
   return true;
}
```

```
// consume next token and return true if it matches pat, else false
public String require(Scanner s, String pat,){
   if ( s.hasNext(pat) ) { s.next(); return true; }
   else { return null; }  // Print error message?
}
```

A Better parser: using patterns

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

```
Pattern numPat = Pattern.compile(
                "[-+]?(\\d+([.]\\d*)?|[.]\\d+)");
Pattern addPat = Pattern.compile("add");
Pattern subPat = Pattern.compile("sub");
Pattern mulPat = Pattern.compile("mul");
Pattern divPat = Pattern.compile("div");
Pattern opPat =
        Pattern.compile("add|sub|mul|div");
Pattern openPat = Pattern.compile("\\(");
Pattern commaPat = Pattern.compile(",");
Pattern closePat = Pattern.compile("\\)");
// Should all be declared as private and final.
```

A Better parser: using patterns

```
public Node parseExpr(Scanner s) {
  Node n;
  if (!s.hasNext())      { return false; }
  if (s.hasNext(numPat)) { return parseNumber(s); }
  if (s.hasNext(addPat)) { return parseAdd(s); }
  if (s.hasNext(subPat)) { return parseSub(s); }
  if (s.hasNext(mulPat)) { return parseMul(s); }
  if (s.hasNext(divPat)) { return parseDiv(s); }
  return false;
}
```