COMP261 Parsing 2 of 4

Marcus Frean

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

CAPITAL CITY UNIVERSITY

today

- the basic idea of a grammar
- an example: the grammar for HTML
- but first: need to break the input into tokens (with a regex)
- the full parse tree *versus* Abstract Syntax Tree
- Top Down Recursive Descent parsing – the basic idea

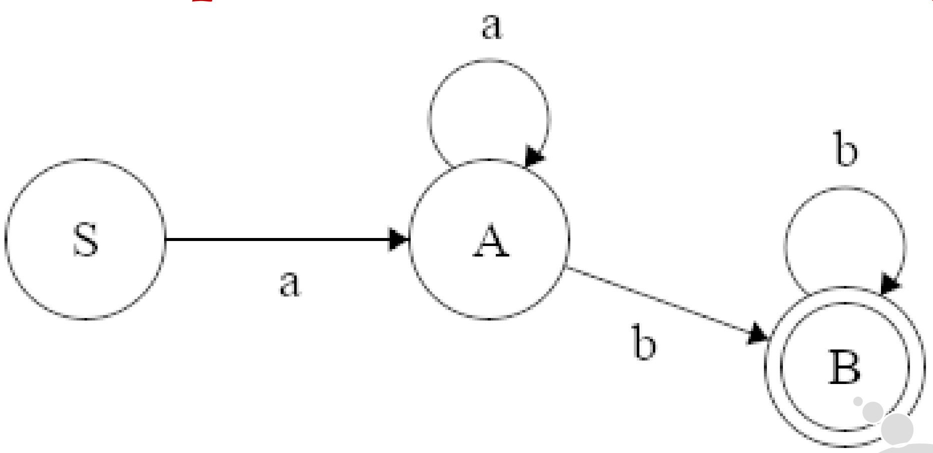
plus… (not examinable)

- regex *versus* this kind of grammar
  - Regex are "regular" grammars, but those can't do nesting
  - visual argument for why

Regex and Acceptors

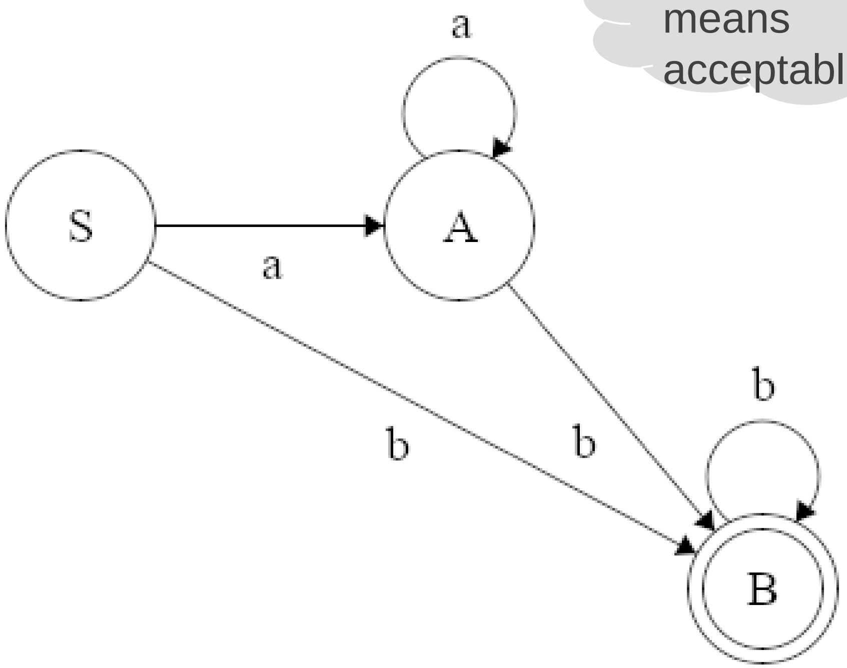
(nb – this stuff about acceptors is not examinable)

- a+b+
- aa*bb*



double circle means acceptable

- a*b+



optional: we could have just made A the start state…

ways of representing a language #2: **grammar**

Rules that spell out what's possible, eg:

- S → aA
- A → aA | bB
- B → bB | end

(this is same as the regex: a+b+)

[Interest only: regex's correspond to just the simplest grammars - the "regular" ones]

A 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 >
```

Nonterminals

- elements of the grammar that are not part of the text
- defined by rules

Top level nonterminal 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 >
```

Just as with regex’s:

|	= "or"
[*NT*]	= "optional"
[*NT*]*	= "any number of times"
[*NT*]+	= "one or more times"

Terminals

- literal strings or patterns of characters

```
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 >
```

parsing text from raw input

Given some text, and a grammar

First we have to…

- break up text into a sequence of tokens ("Lexing")
  - this obviously has to "fit" the language in question

And then we can use the grammar to…

- Parse that token sequence, which could mean:
  - (a) check if the text meets the grammar rules, or
  - (b) construct the **parse tree** for the text

breaking the input into tokens

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:

- Still 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 <u>not be returned in tokens</u>

eg: breaking the input into tokens

We can use a Scanner with a purpose-built delimiter, like this:

```
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");

}
```

This is a complex regex! (details not examinable)
This one works with the HTML grammar ☺

- spaces are separator characters and not part of the tokens
- tokens also delimited at < and > but these are being left in the tokens ☹
- cf. assignment 4 uses \\s+|(?=[{}(),;])|(?<=[{}(),;])

scan.useDelimiter("\\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>
```

- the scanner would generate the tokens:

```
<html>
<head>
<title>
Something
</title>
</head>
<body>
<h1>
```

and so on………

Aside, on Lexical Analysis

Defining delimiters can get very tricky

- Some languages (such as lisp, html, xml) are designed to be easy.

A better approach:

- Define a pattern matching the *tokens* (instead of a pattern matching the *separators*)
- 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
- The patterns can be regular expressions.
  ⇒ use an Acceptor automaton to match / recognise them.

There are tools to make this easier:
see http://en.wikipedia.org/wiki/Lexical_analysis

Now let's use the Grammar to "parse" some text

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?

A 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 >
```

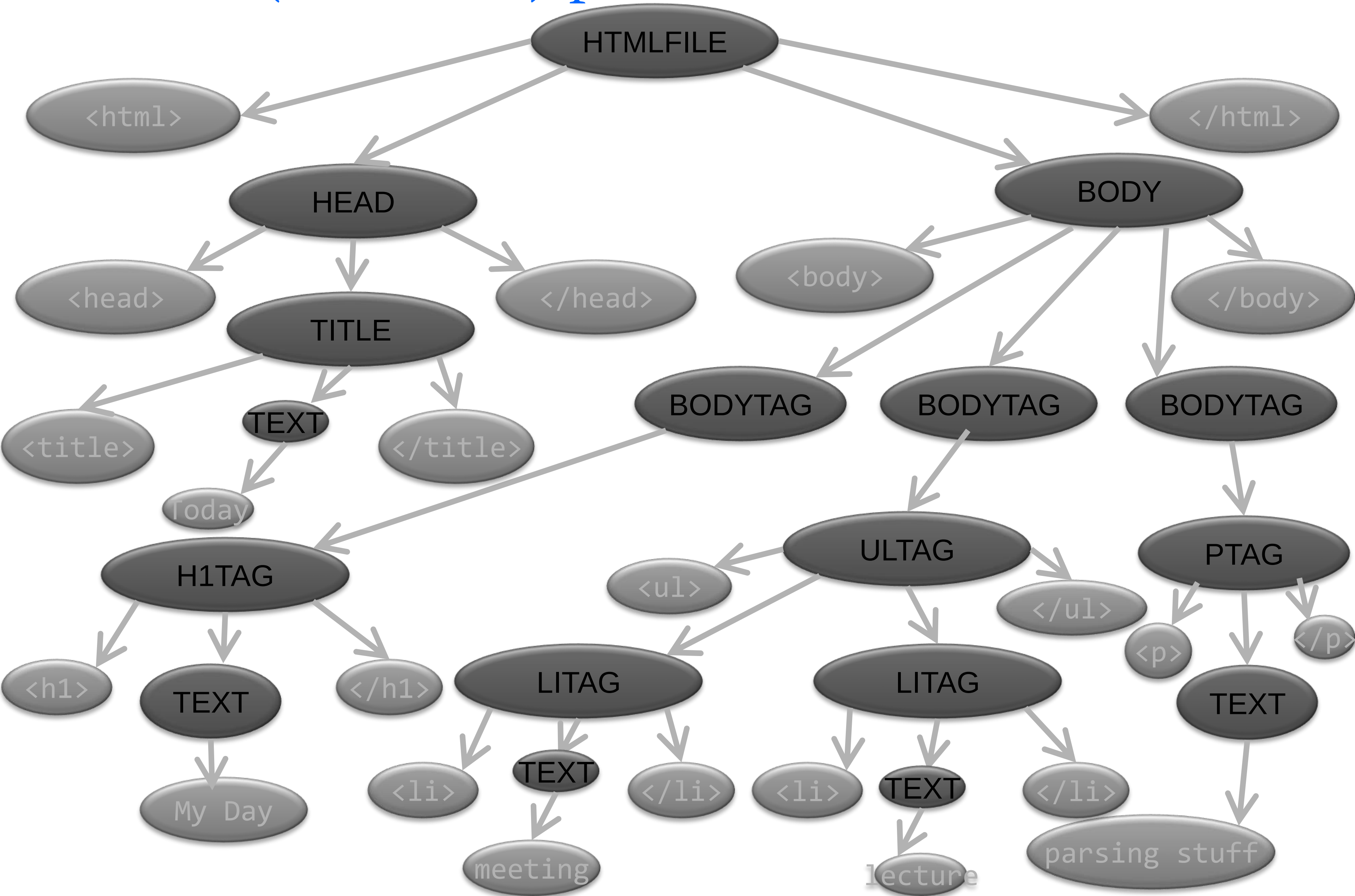
What kind of structure?

- Text conforming to a grammar has a tree structure
  - Ordered tree – order of the children matters
  - Each node in the tree and its children correspond to a grammar rule
  - Each internal node labeled by the nonterminal on LHS of rule
  - Leaves correspond to terminals.

- A **concrete parse tree** represents the syntactic structure of a string according to some formal grammar, showing all the components of the rules

- An **abstract syntax tree** leaves out elements of the rules that are not essential to the structure.

The full (‘concrete’) parse tree



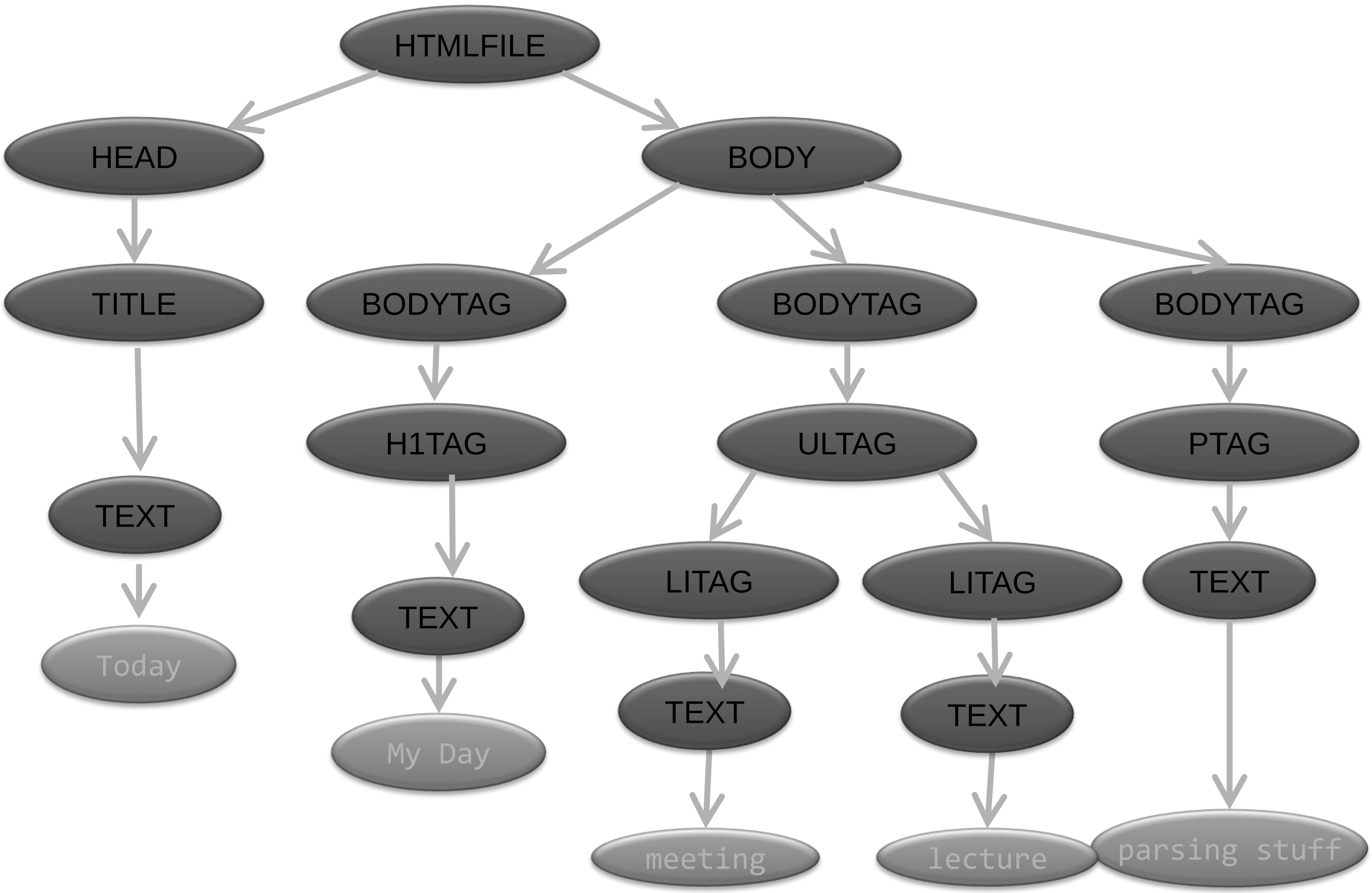
Is that too much information?

- For example, we know that every HEAD will contain "<head>" and "</head>" terminals, we only care about what TITLE there is and only the unknown string part of that title.

→An **abstract syntax tree (AST)** is a tree representation of the abstract syntactic structure of the text.

The syntax in the AST is 'abstract' in that it does not represent everything in the full syntax.

Abstract Syntax Tree (AST)



How do we write programs to do this?

(i.e. to take original text → AST)

Reminder: the process of getting from the input string to the parse tree consists of two steps:

Lexing (lexical analysis):

sequence of characters → sequence of tokens.

- java.util.Scanner can do lexing for us, using regex

Parsing (syntactic analysis):

sequence of tokens → AST

- Assignment will require you to write a recursive descent parser, to be discussed in the next lecture!

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.

We are going to look at a somewhat **naïve** one, which:

- 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 our grammar rules to be highly constrained:

  always able to choose next path given current state and next token

Basic idea: Write a program to mimic 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:

```
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
}
```

Addendum (non-examinable)

Today we've gone beyond regex (== **regular** grammars) to what are called **<u>context free</u>** grammars (CFG).

Regular grammars can't do "nesting", but CFG can.

e.g. can you write a regex for a language that includes

(x) ((x)) (((x))) ((((x)))) ….

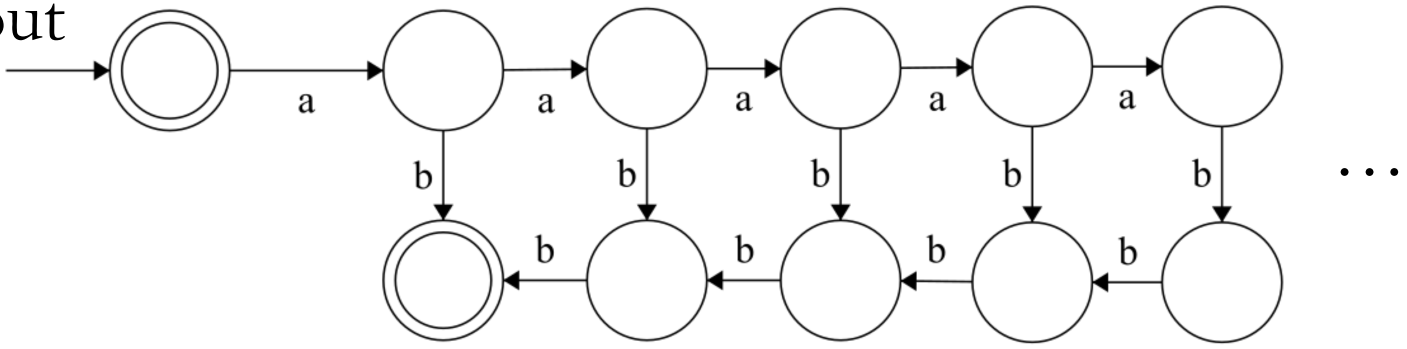
?

Specific cases are fine, but the general matching between number of brackets isn't. This CFG can do it though:

EXPR ::= "x" | "(" EXPR ")"

Addendum (non-examinable)

how about



Intuitively, you'd need a **stack**, right?

This CFG can do it (it's got recursion, hence will use a stack)

EXPR ::= "a" "b" | "a" EXPR "b"

further reading

- here's a Reading (perhaps 30 mins), complete with examples and self-check exercises:

http://web.mit.edu/6.005/www/fa16/classes/17-regex-grammars/