


COMP261 Lecture 16

Parsing 1 of 4: Grammars and Parse Trees



Parsing text

Making sense of "structured text":

- Compiling/analysing programs (javac, g++, Python etc)
- Rendering web sites (Chrome, Mozilla Firefox, etc)
- Processing database queries (PostgreSQL, MySQL, etc)
- Domain specific languages
- Checking grammar in a word processor
- Understanding natural language

A **parser** is a program that reads structured text

What is "structured text"?

```
while ( A[k] != x ) { k++; }
```

```
<html><head><title>My Web Page</title></head>
<body>
<p>Thank you for viewing my silly site!</p>
</body></html>
```

```
DELETE FROM DomesticStudentsFor2017
WHERE mark = 'E';
```

Not "structured text"

I KEEP six honest serving-men (they taught me all I knew);
 Their names are What and Why and When and How and Where and Who.
 I send them over land and sea, I send them east and west;
 But after they have worked for me, I give them all a rest.

I let them rest from nine till five, for I am busy then,
 As well as breakfast, lunch, and tea, for they are hungry men.
 But different folk have different views; I know a person small—
 She keeps ten million serving-men, who get no rest at all!

She sends 'em abroad on her own affairs,
 From the second she opens her eyes—
 One million Hows, two million Wheres,
 And seven million Whys!

The Elephant's child, Rudyard Kipling

What is “structured text”?

We must be able to describe the structure of texts:

- A *URL* is a string of the form `http://name/name/...` where each name is a string of letters, digits and dots.
- An *email address* is a string of the form `id@id.id...` where each id is a string of letters and digits.
- A *while statement* is a string of the form `while (exp) stmt` where *exp* is a valid expression and *stmt* is a valid statement
- A *statement* is either a *while statement*, an *if statement*,

What is “structured text”

We must also be able to describe the structure of a particular text so we can extract its components.

- <http://ecs.vuw.ac.nz/lindsay/home.html>
- lindsay@ecs.vuw.ac.nz
- `while (A[k] != x) { k++; }`

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 from the language’s alphabet, and names its structural components.
`whileStmt ::= “while” “(“ condition “)” statement`
- Can show alternative forms:
`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 [ “else” statement ] |
    “while” “(“ exp “)” statement |
    “do” statement “while” “(“ exp “)” |
    “{“ [ statement ]* “}” |
    ...
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 >

```

Nonterminals

- Names of 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 >

```

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

Terminals

- Literal strings or patterns of characters that can occur in texts
- Here they are enclosed in double quote marks

```

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>

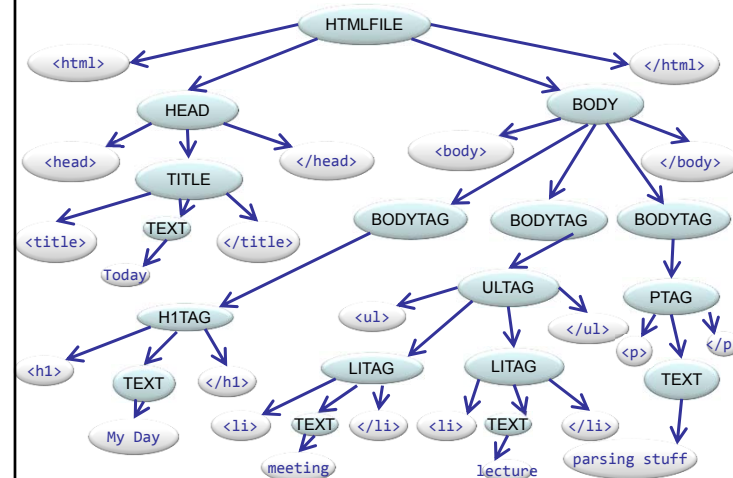
```

- 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 node 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 the terminals on the **fringe** of the tree
- A **concrete syntax tree** or **parse tree** represents the syntactic structure of a string according to some formal grammar, showing all the components of the rules

Concrete Parse Tree



Grammars define possible parse trees

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

H1TAG ::= "<h1>" TEXT "</h1>"

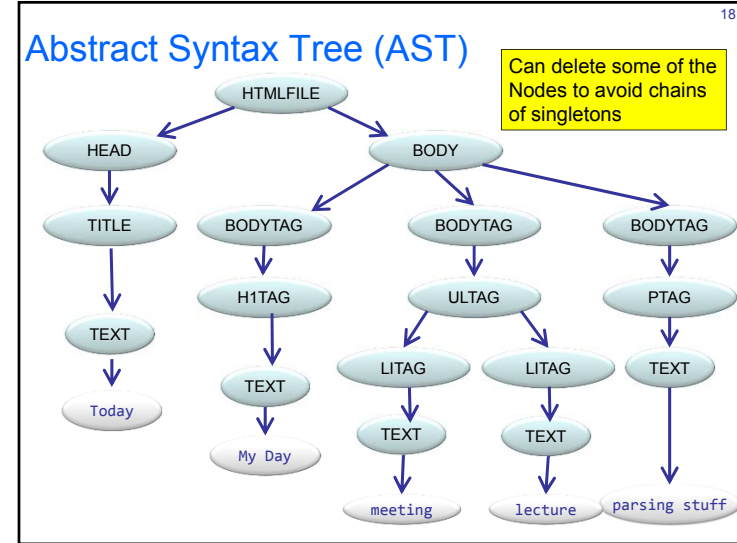
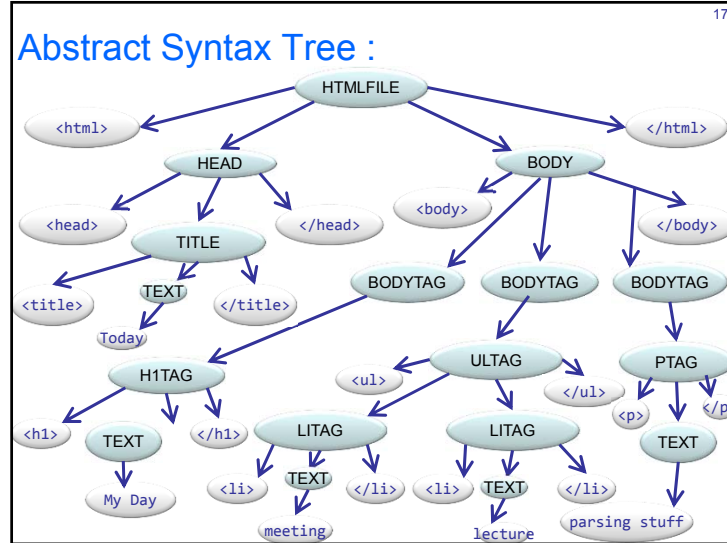
BODYTAG ::= H1TAG | PTAG | OLTAG | ULTAG

HTMLFILE ::= "<html>" [HEAD] BODY "</html>"

Is that too much information?

Yes! We know that every HEAD has “<head>” and “</head>” terminals, we only care about what TITLE there is and only the unknown string part of the title.

- An **abstract syntax tree (AST)** represents the abstract syntactic structure of the text.
- Each node of the tree denotes a construct occurring in the text.
- The syntax is 'abstract' in that it does not represent all the elements of the full syntax.
- Only keep things that are semantically meaningful.



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 a 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!