An Introduction to Grammars and Parsing

Hayo Thielecke
University of Birmingham
www.cs.bham.ac.uk/~hxt

11 October 2010
Outline of the parsing part of the module

1. Intro to grammars using well-bracketing
2. Formal definition of grammars, derivations, and parsers
3. From grammars to parsing methods
4. From grammars to parse trees as data structures
5. Parser generators
Why do you need to learn about grammars?

- Grammars are widespread in programming.
- XML is based on grammars and needs to be parsed.
- Knowing grammars makes it much easier to learn the syntax of a new programming language.
- Powerful tools exist for parsing (e.g., yacc, bison, ANTLR, SableCC, ...). But you have to understand grammars to use them.
- Grammars give us examples of some more advanced object-oriented programming: Composite Pattern and polymorphic methods.
- You may need parsing in your final-year project for reading complex input. Or parser generators later in your job.
Note on examples

I like simple examples. Particularly on slides.
As simple as possible, but no simpler.
There is enough going on (e.g., recursion, correctness by construction).
If you like complication, you can always try:

- The official XML definition
- Construction of LALR parsers

Note: aggressively stripping out unnecessary details is common in CS research.
E.g., λ calculus as a model of programming languages (3 rules).
Grammars and brackets

Grammars are good at expressing various forms of bracketing structure.
Dyck language: all well bracketed strings over some alphabet of brackets.

\[
[ [ ] ] \\
( ) [ { } ] ( [ ] )
\]

But not [ ) or ] [.

Example: module web page structure in XHTML.

```xml
<html>
  <head>
    <title>
      Software Systems Components 1, 2011-2012
    </title>
  </head>
  <body>
    <h1>
      Software Systems Components 1, 2011/2012
    </h1>
  </body>
</html>
```
The Dyck language comes from Mathematics (circa 1882)

Compare: inverses

\[
3 \times \frac{1}{3} \times 5 \times 7 \times \frac{1}{7} \times \frac{1}{5} = 1
\]
The Dyck language comes from Mathematics (circa 1882)

Compare: inverses

$$3 \times \frac{1}{3} \times 5 \times 7 \times \frac{1}{7} \times \frac{1}{5} = 1$$

Matching brackets:

$$( ) [ \{ \} ]$$

For CS+Math: group theory.
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input: Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack: remove ] from the input, pop ] off the stack, carry on
   (b) If a different symbol is on the top of the stack: stop and reject the input

3. If the input is empty:
   (a) If the stack is also empty: accept the input
   (b) If the stack is not empty: reject the input
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   - (a) If the same ] is on the top of the stack: remove ] from the input, pop ] off the stack, carry on
   - (b) If a different symbol is on the top of the stack: stop and reject the input

2. If the input is empty:
   - (a) If the stack is also empty: accept the input
   - (b) If the stack is not empty: reject the input
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack
How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack:
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack:
       remove ] from the input, pop ] off the stack, carry on
How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack:
       remove ] from the input, pop ] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack:
       remove ] from the input, pop ] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
       stop and reject the input
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack:
       remove ] from the input, pop ] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
       stop and reject the input

3. If the input is empty:
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack:
       remove ] from the input, pop ] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
       stop and reject the input

3. If the input is empty:
   (a) If the stack is also empty: accept the input
Dyck language is the killer app for stacks

How can we parse the Dyck language using a stack?

1. If we see a [ in the input:
   Remove [ from the input, push the corresponding ] onto the stack

2. If we see a ] in the input:
   (a) If the same ] is on the top of the stack:
       remove ] from the input, pop ] off the stack, carry on
   (b) If a different symbol is on the top of the stack:
       stop and reject the input

3. If the input is empty:
   (a) If the stack is also empty: accept the input
   (b) If the stack is not empty: reject the input
Intuition: reg exps are like `while` and `if`

A regular expression

\[(a \mid b\ c)^*\]
Intuition: reg exps are like while and if

A regular expression

\[(a | b c)^*\]

while(randomBool()) {
  if(randomBool()) {
    print('a');
  }
  else {
    print('b');
    print('c');
  }
}
Grammars also describe patterns

A context-free grammar for the Dyck language

\[
\begin{align*}
D & \rightarrow [ D ] D \\
D & \rightarrow 
\end{align*}
\]
Grammars also describe patterns

A context-free grammar for the Dyck language

\[ D \rightarrow [ D ] D \]  \hspace{1cm} (1)

\[ D \rightarrow \]  \hspace{1cm} (2)

The grammar generates all well-bracketed strings:

- The empty string
Grammars also describe patterns

A context-free grammar for the Dyck language

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow [D]D$</td>
<td>(1)</td>
</tr>
<tr>
<td>$D \rightarrow ______$</td>
<td>(2)</td>
</tr>
</tbody>
</table>

The grammar generates all well-bracketed strings:

- The empty string
- $[\_\_\_\_]$

---
Grammars also describe patterns

A context-free grammar for the Dyck language

\[
\begin{align*}
D & \rightarrow [D] D \\
D & \rightarrow
\end{align*}
\]

(1)

(2)

The grammar generates all well-bracketed strings:

- The empty string
- [ ]
- [ ] [ ]
- [ ] [ ] [ ]
Grammars also describe patterns

A context-free grammar for the Dyck language

\[
D \rightarrow [D]D \\
D \rightarrow \]

(1)

(2)

The grammar generates all well-bracketed strings:

- The empty string
- [ ]
- [ ] [ ]
- [ [ ] ]
- [ [ ] ]
- [ [ ] [ ] ]
- [ [ ] [ [ ] ] ]
Grammars also describe patterns

A context-free grammar for the Dyck language

\[
\begin{align*}
D & \rightarrow [D] D \\
D & \rightarrow
\end{align*}
\]

The grammar generates all well-bracketed strings:

- The empty string
- [ ]
- [ ] [ ]
- [ ] [ ] [ ]
- [ ] [ [ ] ] [ ]
- [ ] [ [ ] ] [ ] [ ]
Grammars also describe patterns

A context-free grammar for the Dyck language

\[
\begin{align*}
D & \rightarrow [ D ] D \\
D & \rightarrow 
\end{align*}
\] 

The grammar generates all well-bracketed strings:

- The empty string
- \([\ ]\)
- \([\ ][\ ]\)
- \([\ ][\ ][\ ]\)
- \([\ ][\ ][\ ][\ ][\ ]\)
- and so on: there are infinitely many strings that are generated
Intuition: grammars are like recursive methods

A context-free grammar for the Dyck language

\[ D \rightarrow [D]D \]  \hspace{1cm} (1)

\[ D \rightarrow \]  \hspace{1cm} (2)
Intuition: grammars are like recursive methods

A context-free grammar for the Dyck language

\[
D \rightarrow [D]D \\
D \rightarrow 
\]

void D()
{
    switch(randomBool()) {
    case true:
        print('['); D(); print(']'); D(); break;
    case false:
        break;
    }
}
A regular expression for brackets

\[(\[)\star (\])\star\]*

Note: the square bracket [ is escaped as \[. The round parentheses and Kleene star

\((\ldots)\star\)

are not escaped.
Brackets and Regular Expressions

A regular expression for brackets

```
(((\[)*)((\])*))*
```

Note: the square bracket `[` is escaped as `\[`. The round parentheses and Kleene star

```
(...)*
```

are not escaped.

All strings of matching brackets are generated this way

Example: `[ ] [ [ ] ] [ ]`
A regular expression for brackets

```plaintext
(((\[)*)((\])*)*)
```

Note: the square bracket `[` is escaped as `\[`. The round parentheses and Kleene star

```plaintext
(\.*
```

are not escaped.

All strings of matching brackets are generated this way

Example: `[ ] [ [ ] ] [ ]`

But it overgenerates

Example junk: `] ] ] ] [ ]`
Write a program that reads strings like this and evaluates them:

\[(2+3\times(2-3-4))\times2\]

In particular, brackets and precedence must be handled correctly (* binds more tightly than +).

If you attempt brute-force hacking, you may end up with something that is inefficient, incorrect, or both.

This sort of problem was major challenge in the 1950s and 60s in compiling.

Parsing technology makes it straightforward.

Moreover, the techniques scale up to more realistic problems.
What do we need in a grammar

- Some symbols can occur in the actual syntax
- We also need other symbols that act as placeholders
- Rules then say how to replace the placeholders
- E.g. in Java, placeholders include “expression” “statement” etc
A context-free grammar consists of

- some terminal symbols $a, b, \ldots, +, )$, \ldots
- some non-terminal symbols $A, B, S$, \ldots
- a distinguished non-terminal start symbol $S$
- some rules of the form

$$A \rightarrow X_1 \ldots X_n$$

where $n \geq 0$, $A$ is a non-terminal, and the $X_i$ are symbols.
Mathematicians and computer scientists are inordinately fond of Greek letters.

\[ \alpha \quad \text{alpha} \\
\beta \quad \text{beta} \\
\gamma \quad \text{gamma} \\
\varepsilon \quad \text{epsilon} \]
Notational conventions for grammars

- We will use Greek letters \( \alpha, \beta, \ldots \), to stand for strings of symbols that may contain both terminals and non-terminals.
- In particular, \( \varepsilon \) is used for the empty string (of length 0).
- We will write \( A, B, \ldots \) for non-terminals.
- Terminal symbols are usually written in typewriter font, like for, while, [.
- These conventions are handy once you get used to them and are found in most books.
If $A \rightarrow \alpha$ is a rule, we can replace $A$ by $\alpha$ for any strings $\beta$ and $\gamma$ on the left and right:

$$\beta A \gamma \Rightarrow \beta \alpha \gamma$$

This is one derivation step.

A string $w$ consisting only of terminal symbols is generated by the grammar if there is a sequence of derivation steps leading to it from the start symbol $S$:

$$S \Rightarrow \cdots \Rightarrow w$$
An example derivation

Recall the rules

\[
\begin{align*}
D & \rightarrow [D]D \\
D & \rightarrow 
\end{align*}
\]  

(1)  

(2)

There is a unique leftmost derivation for each string in the language. For example, we derive \([ ] [ ]\) as follows:

\[
D
\]
Recall the rules

\[
D \rightarrow [D] D \quad (1)
\]
\[
D \rightarrow \quad (2)
\]

There is a unique leftmost derivation for each string in the language. For example, we derive \([ \] [ ]\) as follows:

\[
D
\Rightarrow [D] D
\]
An example derivation

Recall the rules

\[ D \rightarrow [D] D \]  \quad (1)

\[ D \rightarrow \]  \quad (2)

There is a unique leftmost derivation for each string in the language. For example, we derive \([ ] [ ]\) as follows:

\[ \begin{align*}
D & \Rightarrow [D] D \\
& \Rightarrow [D] D \\
& \Rightarrow [ ] D
\end{align*} \]
An example derivation

Recall the rules

\[ D \rightarrow [D] D \quad (1) \]
\[ D \rightarrow \quad (2) \]

There is a unique leftmost derivation for each string in the language. For example, we derive [ ] [ ] as follows:

\[
\begin{align*}
D & \Rightarrow [D] D \\
& \Rightarrow [ ] D \\
& \Rightarrow [ ] [D] D
\end{align*}
\]
An example derivation

Recall the rules

\[
D \rightarrow [D]D \\
D \rightarrow
\]

(1)

(2)

There is a unique leftmost derivation for each string in the language. For example, we derive [ ] [ ] as follows:

\[
D \\
\Rightarrow [D]D \\
\Rightarrow [ ]D \\
\Rightarrow [] [D]D \\
\Rightarrow [ ] [ ] D
\]
Recall the rules

\[
\begin{align*}
  D & \rightarrow [D] D \\
  D & \rightarrow
\end{align*}
\]

(1) (2)

There is a unique **leftmost** derivation for each string in the language. For example, we derive \([ \ ] [ \ ]\) as follows:

\[
\begin{align*}
  D & \Rightarrow [D] D \\
  & \Rightarrow [ \ ] D \\
  & \Rightarrow [ \ ] [D] D \\
  & \Rightarrow [ \ ] [ \ ] D \\
  & \Rightarrow [ \ ] [ \ ]
\end{align*}
\]
The language of a grammar

- In this context, a *language* is a *set of strings* (of terminal symbols).
- A language is called context-free if it is the language of some context-free grammar.
- For useful grammars, there are usually *infinitely* many strings in its language (e.g., all Java programs).
- Wilhelm von Humboldt wrote (c. 1830) that language “makes *infinite* use of *finite* means”.
- Two different grammars can define the same language. Sometimes we may redesign the grammar as long as the language remains the same.
Two grammars for the same language

Recall

\[ D \rightarrow [D]D \quad (1) \]
\[ D \rightarrow \quad (2) \]

The usual presentation is easier to understand, but not suitable for parsing:

\[ D \rightarrow [D] \]
\[ D \rightarrow D \ D \]
\[ D \rightarrow \]
Grammars can encode regular expressions

The alternative $\alpha | \beta$ can be expressed as follows:
The alternative $\alpha \mid \beta$ can be expressed as follows:

\[
A \rightarrow \alpha \\
A \rightarrow \beta
\]
Grammars can encode regular expressions

The alternative $\alpha | \beta$ can be expressed as follows:

\[
\begin{align*}
A & \rightarrow \alpha \\
A & \rightarrow \beta
\end{align*}
\]

Repetition $\alpha^*$ can be expressed as follows:

\[
\begin{align*}
A & \rightarrow \alpha A \\
A & \rightarrow \\
A & \rightarrow
\end{align*}
\]
Grammars can encode regular expressions

The alternative $\alpha | \beta$ can be expressed as follows:

$$A \rightarrow \alpha$$

$$A \rightarrow \beta$$

Repetition $\alpha^*$ can be expressed as follows:

$$A \rightarrow \alpha A$$

$$A \rightarrow$$

Hence we can use $|$ and $^*$ in grammars; sometimes called BNF, for Backus-Naur-form. Reg exps are still useful, as they are simple and efficient (in grep and lex)
Grammar notations

Warning: notations differ and clash.
In reg exps, brackets are used for character classes, e.g.

\[A–D]\ means A | B | C | D

In BNF, brackets can stand for option:

\[\alpha]\ means \alpha | \varepsilon

Sometimes \{\alpha\} is written for \alpha^*, e.g. in Sun’s Java grammar.
Other notation: \rightarrow\ may be written as “::=” or “.”.
Nesting in syntax

Specifically, we need to be very careful about bracketing and nesting. Compare:

```c
while(i < n)
a[i] = 0;
i = i + 1;
```

and

```c
while(i < n) {
a[i] = 0;
i = i + 1;
}
```

Theses snippets looks very similar. But their difference is clear when indented properly—which requires parsing.
Example from programming language syntax

Some rules for statements $S$ in Java or C:

\[
S \rightarrow \text{if} \ (E) \ S \ \text{else} \ S \\
S \rightarrow \text{while} \ (E) \ S \\
S \rightarrow \ V = E; \\
S \rightarrow \ { } \ B \} \\
B \rightarrow \ S \ B \\
B \rightarrow \\
\]

Here $V$ is for variables and $E$ for expressions.

\[
E \rightarrow \ E \ - \ 1 \\
E \rightarrow (E) \\
E \rightarrow 1 \\
E \rightarrow E \ == \ 0 \\
V \rightarrow \ foo
\]
A Java grammar rule from Oracle

This is one of more than 60 grammar rules for the syntax of Java:

Primary:

( Expression )
this [Arguments]
super SuperSuffix
Literal
new Creator
Identifier { . Identifier }[ IdentifierSuffix]
BasicType BracketsOpt .class
void.class

Note: the colon is like → and a new line is like |
Definition of a parser

Suppose a grammar is given. A parser for that grammar is a program such that for any input string \( w \):

- If the string \( w \) is in the language of the grammar, the parser finds a derivation of the string. In our case, it will always be a leftmost derivation.
- If string \( w \) is \textbf{not} in the language of the grammar, the parser must reject it, for example by raising a parsing exception.
- The parser always terminates.
Recursion in grammars

A symbol may occur on the right hand-side of one of its rules:

\[ E \rightarrow (E) \]

We often have **mutual** recursion in grammars:

\[ S \rightarrow \{ B \} \]
\[ B \rightarrow S B \]

Mutual recursion also exists in Java: for example, method \( f \) calls \( g \) and \( g \) calls \( f \).
Recursion in grammars and in Java

Compare recursion in grammars, methods and classes:

\[ T \rightarrow \ldots T \ldots T \ldots \]

```java
int sumTree(Tree t)
{
    ... return sumTree(t.left) + sumTree(t.right);
}
```

and classes

```java
public class Tree
{
    ... public Tree left;
    public Tree right;
}
```
Automata and grammars—what we are not going to do

- Pushdown automata (stack machines) are covered in Models of Computation.
- Independently of formal automata models, we can use a programming perspective: grammars give us classes or methods.
- For experts: we let Java manage the stack for us by using methods
- The parsing stack is part of the Java call stack
There are many different parsing technologies (LL(k), LR(1), LALR(1), ...).

Here we consider only predictive parsers, sometime also called recursive descent parsers. They correspond to translating grammar rules into code as described below.

The hard part is choosing the rules according to the lookahead.

ANTLR works similarly, but with more lookahead.
A **predictive** parser can be constructed from grammar rules.

The parser is allowed to “look ahead” in the input; based on what it sees there, it then makes predictions.

Canonical example: matching brackets.
If the parser sees a `[ as the next input symbol, it “predicts” that the input contains something in brackets.

More technically: `switch` on the lookahead; `[ labels one case.
Methods for processing the language follow the structure of the grammar:
Each non-terminal gives a method (with mutual recursion between such methods).
Each grammar gives us a Java class hierarchy (with mutual recursion between such classes).
Each word in the language of the grammar gives us an object of the class corresponding to the start symbol (a parse tree).
Recursive methods

From grammars to mutually recursive methods:

- For each non-terminal $A$ there is a method $A$. The method body is a switch statement that chooses a rule for $A$.
- For each rule $A \rightarrow X_1 \ldots X_n$, there is a branch in the switch statement. There are method calls for all the non-terminals among $X_1, \ldots, X_n$.

Each grammar gives us some recursive methods. For each derivation in the language, we have a sequence of method calls.
The lookahead and match methods

- A predictive parser relies on two methods for accessing the input string:
- `char lookahead()` returns the next symbol in the input, without removing it.
- `void match(char c)` compares the next symbol in the output to `c`. If they are the same, the symbol is removed from the input. Otherwise, the parsing is stopped with an error; in Java, this can be done by throwing an exception.
We also need to know where else in the grammar a $D$ could occur:

\[
S \rightarrow D \$ 
\]

Idea: suppose you are trying to parse a $D$. Look at the first symbol in the input:

if it is a [, use the first rule;
if it is a ] or $\$, use the second rule.
void parseD() throws SyntaxError
{
    switch(lookahead()) { // what is in the input?
    case '[': // If I have seen a [
        match('['); // remove the [
        parseD(); // now parse what is inside
        match(']'); // make sure there is a ]
        parseD(); // now parse what follows
        break; // done in this case
    case ']': case '$': // If I have seen a ] or $
        break; // just return
    default: throw new SyntaxError();
    }
}
How do we get the symbols for the case labels?

- Parsing with lookahead is easy if every rule for a given non-terminal starts with a different terminal symbol.

  
  \[
  D \rightarrow [D] D \\
  D \rightarrow (D) D \\
  \ldots
  \]

- In that case, the lookahead immediately tells us which rule to choose.

- But what if not? The right-hand-side could instead start with a non-terminal, or be the empty string.

- More general methods for using the lookahead: FIRST and FOLLOW construction.
FIRST and FOLLOW construction for predictive parsing

\[
\text{FIRST}(\alpha) = \{ b \mid \exists \beta. \alpha \Rightarrow^* b \beta \}
\]

\[
\text{FOLLOW}(X) = \{ b \mid \exists \alpha. \exists \gamma. S \Rightarrow^* \alpha X b \gamma \}
\]
FIRST and FOLLOW explained

We define FIRST, FOLLOW and nullable:

- A terminal symbol $b$ is in $\text{FIRST}(\alpha)$ if there is a derivation
  \[ \alpha \Rightarrow^* b \beta \]
  ($b$ is the first terminal symbol in something derivable from $\alpha$).
- A terminal symbol $b$ is in $\text{FOLLOW}(X)$ if if there is a derivation
  \[ S \Rightarrow^* \alpha X b \gamma \]
  ($b$ can appear immediately behind $X$ in some derivation)
- $\alpha$ is nullable if $\alpha \Rightarrow^* \varepsilon$ (we can derive the empty string from it)
FIRST and FOLLOW give the case labels for the branches of the switch statement.

A branch for $A \rightarrow \alpha$ gets the labels in $\text{FIRST}(\alpha)$.

A branch for $A \rightarrow \varepsilon$ gets the labels in $\text{FOLLOW}(A)$.

FIRST and FOLLOW are tedious to compute by hand. We won’t go into the details here.

Parser generators like ANTLR compute this sort of information automagically.
FIRST and FOLLOW guide the Dyck language parser

The parser is a method constructed from the grammar rules:

\[
D \rightarrow [ D ] D \\
D \rightarrow 
\]

In this case, we have

\[
\text{FIRST}( [ D ] D ) = \{ [ ] \} \\
\text{FOLLOW}(D) = \{ [ ], $ \}
\]

Intuition:

**FIRST** tells us that if we see a [, we use rule (1)

**FOLLOW** tells us that if we see a ] or the end of input $, we use rule (2)
Suppose we have two rules for the same $A$:

\[
\begin{align*}
A & \rightarrow \alpha \\
A & \rightarrow \beta
\end{align*}
\]

such that there is some $a$ in both $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$
Suppose we have two rules for the same $A$:

\begin{align*}
A & \rightarrow \alpha \\
A & \rightarrow \beta
\end{align*}

such that there is some $a$ in both \text{FIRST}(\alpha) and \text{FIRST}(\beta).

If we see $a$ in the lookahead, we must use both rules: conflict.
This is another grammar for the Dyck language:

\[
\begin{align*}
D & \rightarrow \ [ \ D \ ] \\
D & \rightarrow \ D \ D \\
D & \rightarrow \\
\end{align*}
\]

This grammar is nice and clean: one rule each for bracketing, sequential composition, and empty string.
This is another grammar for the Dyck language:

\[
D \rightarrow [ \, D \, ] \\
D \rightarrow D \, D \\
D \rightarrow
\]

This grammar is nice and clean: one rule each for bracketing, sequential composition, and empty string.

But: \textsc{first} for the first two rules both contains [ Special case of FIRST/FIRST conflict: \textit{left} recursion. Thus this grammar is no good for predictive parsing.
If we can parse the Dyck language, we can parse (at least the core of) XHTML. However, we need to treat a whole tag as if it were a single symbol

\[ ... \]

If the lookahead sees just a `<`, it will not work. Must see the whole `<title>`
This is done by preprocessing the input using a scanner. Stream of tokens, not individual chars.
The scanner tells the parser: now we see a `<title>`, now `<p>`, now we see `</title>`, ...
Example: parsing [ ] $ 

\[
D \rightarrow [D]D \\
D \rightarrow \ \\
\]

1 Call parseD
Example: parsing \([ \ ] \) $$

\begin{align*}
D & \rightarrow [D]D \\
D & \rightarrow
\end{align*}

1. Call parseD
2. lookahead returns [, thus switch to rule (1)
Example: parsing \[ \] \$

\[
D \rightarrow [D] D \quad (1)
\]
\[
D \rightarrow \quad (2)
\]

1. Call \texttt{parseD}
2. \texttt{lookahead returns [}, thus switch to rule (1)
3. \texttt{match [}
Example: parsing \[ ] \$ \\

\[
D \rightarrow [D]D \quad (1)
\]
\[
D \rightarrow \quad (2)
\]

1. Call \texttt{parseD}
2. lookahead returns \[,\] thus switch to rule (1)
3. match [\
4. Call \texttt{parseD} recursively for the first \(D\)
Example: parsing \([ \ ]\) \$ 

\[
D \rightarrow [D] D \\
D \rightarrow 
\]

(1) \[
D \rightarrow [D] D
\]

(2) \[
D \rightarrow
\]

1. Call \text{parseD}
2. \text{lookahead} returns [, thus switch to rule (1)
3. match [ 
4. Call \text{parseD} recursively for the first \(D\) 
5. \text{lookahead} returns ], so return immediately by (2)
Example: parsing \[ \] $ \\

$ D \rightarrow [ D ] D $(1)$  \\
$ D \rightarrow $(2)$  \\

1 Call parseD  
2 lookahead returns $[, thus switch to rule (1)$  
3 match $[  
4 Call parseD recursively for the first $D$  
5 lookahead returns $]$, so return immediately by (2)  
6 match $]$
Example: parsing \[ ] \$ \\

\[ D \rightarrow [D]D \] \hspace{1cm} (1) \\
\[ D \rightarrow \] \hspace{1cm} (2) \\

1. Call \texttt{parseD} \\
2. \texttt{lookahead} returns [, thus switch to rule (1) \\
3. \texttt{match [} \\
4. Call \texttt{parseD} recursively for the first \texttt{D} \\
5. \texttt{lookahead returns ]}, so return immediately by (2) \\
6. \texttt{match ]} \\
7. Call \texttt{parseD} recursively for the second \texttt{D}
Example: parsing $[ ]$ 

\[
D \rightarrow \ [ D ] \ D \quad (1)
\]
\[
D \rightarrow \quad (2)
\]

1. Call \texttt{parseD}
2. \texttt{lookahead} returns $[$, thus switch to rule (1)
3. \texttt{match [}
4. Call \texttt{parseD} recursively for the first $D$
5. \texttt{lookahead} returns $]$, so return immediately by (2)
6. \texttt{match ]}
7. Call \texttt{parseD} recursively for the second $D$
8. \texttt{lookahead} returns $\$$, so return immediately by (2)

\text{return: parsing was successful with only $\$$ left}
Example: parsing [ ] $

\[
\begin{align*}
D & \rightarrow [ D ] D \\
D & \rightarrow
\end{align*}
\]  \quad (1)  

1. Call parseD
2. lookahead returns [, thus switch to rule (1)
3. match [ 
4. Call parseD recursively for the first D
5. lookahead returns ], so return immediately by (2)
6. match ] 
7. Call parseD recursively for the second D
8. lookahead returns $, so return immediately by (2)
9. return: parsing was successful with only $ left.
Many constructs start with a keyword telling us immediately what it is. Keywords “if”, “while”, produce tokens that tell the parser to expect a conditional, a loop, etc
⇒ these symbols are typically in FIRST
Many constructs end with a terminator like “;”
Such tokens tell the parser to stop reading an expression, statement etc
⇒ these symbols are typically in FOLLOW
Parsers and parse trees

- Parse trees as such are abstract mathematical structures
- We can represent them as data structures
- Many modern tools give you such representations of parse trees
- ANTLR and SableCC build trees, yacc does not
- XML: DOM tree parsers versus streaming XML parsers
- We will see how to represent them in an OO way
- These classes give return types for the parsing methods
Parse trees abstractly

The internal nodes are labelled with nonterminals. If there is a rule \( A \rightarrow X_1 \ldots X_n \), then an internal node can have the label \( A \) and children \( X_1, \ldots, X_n \). The root node of the whole tree is labelled with the start symbol. The leaf nodes are labelled with terminal symbols or \( \varepsilon \).

![Parse Tree Diagram]

Root: Start symbol

- Non-terminal \( A \)
  - Terminal \( a \)
  - \( \cdots \)

- Non-terminal \( B \)
  - \( \varepsilon \)
  - \( \cdots \)
  - Terminal \( z \)
Example: parse trees

\[
D \rightarrow [D]D \\
D \rightarrow \\
\]

Here is a parse tree for the string [ ]:
Parse trees and derivations

- Parse trees and derivations are related, but not the same.
- Intuition: Parse trees are extended in space (data structure), derivations in time.
- For each derivation of a word, there is a parse tree for the word.
  (Idea: each step using $A \rightarrow \alpha$ tells us that the children of some $A$-labelled node are labelled with the symbols in $\alpha$.)
- For each parse tree, there is a (unique leftmost) derivation.
  (Idea: walk over the tree in depth-first order; each internal node gives us a rule.)
Parse tree traversal and derivation
Parse tree traversal and derivation

\[
D
\xrightarrow{[D]} [D] D
\]
Parse tree traversal and derivation

\[
\begin{align*}
    D & \rightarrow [D] D \\
    & \Rightarrow [D] D \\
    & \Rightarrow [ ] D
\end{align*}
\]
Parse tree traversal and derivation

\[ D \Rightarrow [ D ] D \]
\[ \Rightarrow [ ] D \]
\[ \Rightarrow [ ] \]
Abstract characterization of parsing as an inverse

Printing then parsing: same tree.

\[
\text{Tree} \xrightarrow{\text{toString}} \text{String} = \text{parse} \downarrow \text{Tree} \cup \text{SyntaxError}
\]

Parsing then printing: almost the same string.

\[
\text{String} \xrightarrow{\text{parse}} \text{Tree} \cup \text{SyntaxError} \xrightarrow{\text{prettyPrint}} \text{String}
\]
We translate a grammar to some mutually recursive Java classes:

- For each non-terminal $A$ there is an abstract class $A$
- For each rule $A \rightarrow X_1 \ldots X_n$, there is a concrete subclass of $A$.
- It has instance fields for all non-terminals among $X_1, \ldots, X_n$.

(Instead of an abstract class, we could also use an interface for each non-terminal $A$.)
In the grammar, the rules do not have names. We could pick names, or just number the rules for each non-terminal $A$ as $A_1$, ... $A_n$.

Example:

$$D \rightarrow [D]D \quad (1)$$

$$D \rightarrow \quad (2)$$

Then we translate this to two subclasses of class $D$, with constructors

$$D_1(D \ x, D \ y)$$

$$D_2()$$
Method `toString` in the tree classes

The `toString` method of the class for rule

\[ A \rightarrow X_1 \ldots X_n \]

concatenates the `toString` of all the \( X_i \):
- if \( X_i \) is a terminal symbol, it is already a string;
- if \( X_i \) is a non-terminal, its `toString` method is called.

Calling `toString` on a parse tree prints the word at the leaves of the tree.
Example: Dyck language constructors

abstract class D {
    public abstract String toString();
}

class D2 extends D
    // D ->
    {
    D2() {
    public String toString() { return ""; }
    }
}
class D1 extends D
// D -> [ D ] D
{
    private D left, right;

    D1(D l, D r) {
        left = l; right = r;
    }

    public String toString() {
        return "[ " + left.toString() + " ] " + right.toString();
    }
}
For the parser that only recognizes, we had a `void` return type. We extend our translation of grammar rules to code:

- The method for non-terminal A has as its return type the abstract class that we created for A.
- Whenever we call a method for a non-terminal, we remember its return value in a local variable.
- At the end of the translation of a rule, we call the constructor.
D parseD() throws SyntaxError {
    D left, right, result = null;
    switch(lookahead()) {
        case '[':
            match('[');
            left = parseD();
            match(']');
            right = parseD();
            result = new D1(left, right);
            break;
        case ']': case '$':
            result = new D2();
            break;
        default: throw new SyntaxError();
    }
    return result;
}
A grammar is **ambiguous** if there is a string that has **more than one parse tree**.

**Standard example:**

\[
E \rightarrow E - E \\
E \rightarrow 1
\]

One such string is \(1-1-1\). It could mean \((1-1)-1\) or \(1-(1-1)\) depending on how you parse it.

Ambiguous grammars are a problem for parsing, as we do not know which tree is intended.

Note: do not confuse ambiguous with FIRST/FIRST conflict.
Left recursion

In fact, this grammar also has a FIRST/FIRST conflict.

\[
E \rightarrow E - E \\
E \rightarrow 1
\]

1 is in FIRST of both rules
⇒ predictive parser construction fails
Standard solution: left recursion elimination
Left recursion elimination example

\[
E \rightarrow E - E \\
E \rightarrow 1
\]

We observe that \( E \Rightarrow^* 1 - 1 - \ldots - 1 \)

Idea: 1 followed by 0 or more “ - 1”

\[
E \rightarrow 1 F \\
F \rightarrow - 1 F \\
F \rightarrow
\]

This refactored grammar also eliminates the ambiguity. Yay.
This grammar has a FIRST/FIRST conflict

\[ A \rightarrow a \ b \]
\[ A \rightarrow a \ c \]

No left recursion.
No ambiguity.
Note: the regular expression

\[ (a \ b) | (a \ c) \]

is fine.
Reg exps matcher do not depend on lookahead.
This grammar has a FIRST/FIRST conflict

\[ A \rightarrow a \ b \]
\[ A \rightarrow a \ c \]

No left recursion.
No ambiguity.
Note: the regular expression

\[(a \ b) \mid (a \ c)\]

is fine.
Reg exps matcher do not depend on lookahead.
This grammar has a FIRST/FIRST conflict

\[ A \rightarrow a \ b \]
\[ A \rightarrow a \ c \]

No left recursion.
No ambiguity.

Note: the regular expression

\((a \ b) \mid (a \ c)\)

is fine.

Reg exps matcher do not depend on lookahead.
This grammar has a FIRST/FIRST conflict

\[
A \rightarrow a \ b \\
A \rightarrow a \ c
\]

No left recursion.
No ambiguity.
Note: the regular expression

\[(a \ b) | (a \ c)\]

is fine.

Reg exps matcher do not depend on lookahead.
This grammar has a FIRST/FIRST conflict

\[ A \rightarrow a \ b \]
\[ A \rightarrow a \ c \]

No left recursion.
No ambiguity.
Note: the regular expression

\[ (a \ b) \mid (a \ c) \]

is fine.
Reg exps matcher do not depend on lookahead.
The representation of parse trees is an instance of the **COMPOSITE** pattern in object-oriented software. Gamma et al. define it as:

“*Compose objects into tree structures to represent part-whole hierarchies.*”

(From Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.)

This is essentially the representation *parse trees* described above. It describes a “hierarchy” as a tree, where the nodes are the whole and its children the parts.
Consider the binary tree grammar:

\[ B \rightarrow B \ B \mid 1 \mid 2 \mid \ldots \]

In UML notation, we have the following class diagram:

A \( B \) “is a” Leaf or a Node, and a Node “has a” \( B \).
Modern functional languages such as Haskell and ML (with its dialects OCAML and F#) have much better constructs for representing trees. E.g.,

```plaintext
type tree =
  | Leaf of int
  | Node of tree * tree
```
To perform useful work after the parse tree has been constructed, we call its methods.

Canonical example: expression evaluation. Suppose we have a grammar for arithmetical expressions:

\[
E \rightarrow E + E \\
E \rightarrow E \times E \\
E \rightarrow 1 \mid 2 \mid 3 \mid \ldots
\]

The each expression “knows” how to evaluate itself (depending on whether it is an addition, a multiplication, \ldots). The methods in the parse tree can be given additional parameters to move information around the tree.
abstract class Expression {
    abstract int eval() ...
}

class Plus extends Expression {
    ....
    public int eval() {
        return left.eval() + right.eval();
    }
    ....
}
Suppose we also have variables: $E \rightarrow x \mid \ldots$.

```java
abstract class Expression {
    abstract int eval(Environment env);
}

class Variable extends Expression {
    private String name;

    public int eval(Environment env) {
        return env.get(name);
    }
}
```
Parse trees are an instance of the composite pattern.
The classes made for rules $A \rightarrow \alpha$ extend the class for $A$
The return type of a parsing method for $A$ is the abstract class $A$, but what is actually return is an instance of one of its subclasses.
The methods in the abstract class are overridden by the subclasses. During a treewalk, we rely on dynamic dispatch.
Polymorphism is crucial.
The parsing methods could all be static (like C functions), but it could also be done in a more OO style.
Except when the grammar is very simple, one typically does not program a parser by hand from scratch. Instead, one uses a parser generator.

Compare:

```
Java code \[\rightarrow\] Compiler \[\rightarrow\] Java byte code
```

```
Annotated grammar \[\rightarrow\] Parser generator \[\rightarrow\] Parser
```

Examples of parser generators: yacc, bison, ANLTR, JavaCC, SableCC, …
Parsers and lexers

In the real world, \texttt{lookahead} and \texttt{match} are calls to the lexical analyzer (lexer) that matches \textit{regular expressions}. \texttt{lookahead} returns tokens, not raw characters.

Example: substring \texttt{4711666.42} in the input becomes a single token \texttt{FLOAT}.

We ignored all that to keep the parser as simple as possible. (Only single-letter keywords.)

But our simple predictive parsers are sufficient to demonstrate the principles.
More on parser generators

- Parser generators use some ASCII syntax rather than symbols like →.
- With yacc, one attaches parsing actions to each production that tell the parser what to do.
- Some parsers construct the parse tree automatically. All one has to do is tree-walking.
- Parser generators often come with a collection of useful grammars for Java, XML, HTML and other languages.
- If you need to parse a non-standard language, you need a grammar suitable for input to the parser generator.
- Pitfalls: ambiguous grammars, left recursion
Yacc parser generator

- "Yet Another Compiler-Compiler"
- An early (1975) parser generator geared towards C.
- Technically, an LALR(1) parser. LALR(1) is too complicated to explain here.
- Very influential. You should have heard about if for historical reasons.
- Still widely referred to: “This is the yacc for ⟨blah⟩” means “This largely automates doing ⟨blah⟩ while hiding much of the complexity”
- Linux version is called bison, see http://www.gnu.org/software/bison/.
ANTLR parser generator

- Works with Java or C
- Download and documentation at http://antlr.org
- uses LL(k): similar to our predictive parsers, but using more than one symbol of lookahead
- Parse tree can be constructed automatically
- you just have to annotate the grammar to tell ANTLR when to construct a node
- The parse trees are a somewhat messy data structure, not very typed or OO
- ANTLR has been used in several student projects here
ANTLR syntax example

Example of an annotated grammar rule from ANTLR website tutorial:

```antlr
expr returns [int value]
    :   e=multExpr {$value = $e.value;}
        ( '+' e=multExpr {$value += $e.value;}
         |   '-' e=multExpr {$value -= $e.value;}
        )*
    ;
```

In grammar notation:

```
E → M (+ M | − M)*
```

The ANTLR code above does not build a parse tree, but evaluates the expression to an int while parsing.
JavaCC is a parser generator aimed at Java.
See https://javacc.dev.java.net/ for downloads and documentation.
uses LL(1) if possible
Blurb: “Java Compiler Compiler [tm] (JavaCC [tm]) is the most popular parser generator for use with Java [tm] applications.”
SableCC parser generator

- Works with Java
- Download and documentation at http://sablecc.org
- uses LALR(1) for parsing, just like Yacc
- SableCC has no problem with left recursion, as LR parsing does not only depend on the look-ahead
- you may get cryptic errors about shift/reduce and reduce/reduce conflicts if the grammar is unsuitable
- SableCC constructs an object-oriented parse tree, similar to the way we have constructed Java classes
- uses the Visitor Pattern for processing the parse tree
- SableCC has been used in several students projects here
Visitor pattern for walking the parse tree

- Having to modify the methods in the tree classes is poor software engineering.
- It would be much better if we had a general-purpose tree walker into which we can plug whatever functionality is desired.
- The canonical way to do that is the Visitor Design pattern.
- The tree is separate from specialized code that “visits” it.
- The tree classes only have “accept” methods for visitor objects.
- See the *Design Patterns* book by Gamma, Helms, Johnson and Vlissides, or www.sablecc.org.
## Parser generator overview

<table>
<thead>
<tr>
<th>Parser generator</th>
<th>LR or LL</th>
<th>Tree processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yacc/Bison</td>
<td>LALR(1)</td>
<td>Parsing actions in C</td>
</tr>
<tr>
<td>SableCC</td>
<td>LALR(1)</td>
<td>Visitor Pattern in Java</td>
</tr>
<tr>
<td>ANTLR</td>
<td>LL(k)</td>
<td>Tree grammars + Java or C++</td>
</tr>
<tr>
<td>JavaCC</td>
<td>LL(k)</td>
<td>JJTree + Visitors in Java</td>
</tr>
</tbody>
</table>
Summary for parsing

- We have seen some basics of parsing (grammars, derivations, parse trees).
- We have translated grammars to Java code.
- We have touched upon some more advanced material (FIRST/FOLLOW, parser generators, Visitor pattern).
- The code contains generally useful programming concepts: mutual recursion of methods and data, composite pattern, abstract classes, polymorphism, exceptions, . . .
- In practice, one would use a parser generator rather than reinvent the wheel
- One still needs to have some idea of what a parser generator does, as it is not just a button to click on
Books and further reading on parsing

I hope these slides are detailed enough, but if you want to dig deeper:
There are lots of book on compilers. The ones which I know best are:

- Appel, *Modern Compiler Design in Java*.
- Aho, Sethi, and Ullman, nicknamed “The Dragon Book”

Parsing is covered in both, but the Composite Pattern for trees and Visitors for tree walking only in Appel.
See also the websites for ANTLR (http://antlr.org) and SableCC (http://sablecc.org).