L19: Regular expressions & Grammars

Today
1. Grammars
2. Parser-generators
3. Regular expressions

Reading Input
Input to a program is usually represented as a sequence of bytes or a sequence of characters. That input might come from

- A file on disk (in which case the specification is the *file format*)
- A message sent over a network (the specification is a *wire protocol*)
- A command typed by the user on the console (the specification is a *command line interface*)

In today’s lecture, we’re going to talk about how to write the specifications for input using a *grammar*, and how to implement a grammar using a tool called a parser-generator.

We’ll also talk about a specialized form of a grammar called a *regular expression*, which is useful for many string-processing tasks.

Markup and HTTP
For today’s working examples of file formats, we’ll be using two different *markup languages*, which represent typographic style in plain text. Here they are:

**HTML**

Here is an `<i>`italic</i>` word.

**Markdown**

This is _italic_.

For simplicity, our HTML and Markdown grammars will only specify italics.

For another example, which will help illustrate the kind of grammar that would be used for a wire protocol or a command line interface, we’re going to look at URLs:

**URL**

http://web.mit.edu/6.005/www/sp14/psets/ps3/

Grammars
To describe a file format or a wire protocol, we will use a compact representation called a *grammar*. A grammar defines a set of *sentences* (valid strings), where each sentence is a sequence of symbols called *tokens* (also called *terminals*).
For example, our grammar for HTTP will specify the set of sentences that are legal messages in the HTTP protocol.

We'll generally write terminals in quotes, like ‘GET’.

A grammar is a set of productions, where each production defines a nonterminal. You can think of a nonterminal like a variable that stands for a set of sentences.

We’ll conventionally write nonterminals as lowercase identifiers, like x or y or url.

A production has the form

\[
\text{nonterminal ::= expression of terminals, non-terminals and operators}
\]

The three most important operators in a production expression are:

- **Sequence**
  \[
x ::= y z \quad \text{an } z \text{ is a } y \text{ followed by a } z
\]

- **Repetition**
  \[
x ::= y^* \quad \text{an } x \text{ is zero or more } y\text{'s}
\]

- **Choice**
  \[
x ::= y \mid z \quad \text{an } x \text{ is a } y \text{ or a } z
\]

You can also use additional operators which are just syntactic sugar (equivalent to combinations of the big three operators):

- **option**: \[
x ::= y? \quad \text{an } x \text{ is a } y \text{ or is the empty sentence}
\]

- **grouping**: \[
x ::= (y z)^* \quad \text{parentheses for grouping}
\]
  \[
  \text{an } x \text{ is zero or more } y-z \text{ pairs}
\]

- **1+ repetition**: \[
x ::= y^+ \quad \text{is equivalent to } x ::= y y^*
\]
  \[
  \text{an } x \text{ is one or more } y\text{'s}
\]

- **character classes**: \[
x ::= [abc] \quad \text{is equivalent to } x ::= \text{‘a’ | ‘b’ | ‘c’}
\]
  \[
x ::= [^b] \quad \text{is equivalent to } x ::= \text{‘a’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ... (all other characters)}
\]

**Example: URL**

Let’s start with a simple example. Suppose we want to write a grammar that represents URLs. Let’s build up a grammar gradually by starting with simple examples and extending it as we go.

Here’s a simple URL:

http://mit.edu/

A grammar that represents the set of sentences containing only this URL would look like:

\[
\text{url ::= ‘http://mit.edu/’}
\]

But let’s generalize it to capture other domains, as well:

http://stanford.edu/

http://google.com/

We can also write this as one line, like this:

\[
\text{url ::= ‘http://’ [a-z]+ ‘.’ [a-z]+ ‘/’}
\]
In this one-line form, a grammar is called a **regular expression**. But it will be easier to understand if we name the parts using new nonterminals:

```
url ::= ‘http://’ hostname ‘/
hostname ::= identifier ‘.’ identifier
identifier ::= [a-z]+
```

How else do we need to generalize? Hostnames can have more than two components, and there can be an optional port number:

```
http://didit.csail.mit.edu:4949/
```

So the grammar is now:

```
url ::= ‘http://’ hostname (‘:’ port) ‘/
hostname ::= identifier ‘.’ hostname | identifier ‘.’ identifier
port ::= [0-9]+
identifier ::= [a-z]+
```

Notice how **hostname** is now defined recursively in terms of itself. Which part of the hostname definition is the base case, and which part is the recursive step? What possible hostnames are allowed?

Using the repetition operator, we could also write hostname like this:

```
hostname ::= (identifier ‘.’)+ identifier
```

Another thing to observe is that this grammar allows port numbers that are not technically legal. Port numbers can only range from 0 to 65535. We could write a more complex definition of port that would allow only these integers, but that's not typically done in a grammar. Instead,

There are more things we should do to should go farther:

- generalizing ‘http’ to support the additional protocols that URLs can have
- generalizing the ‘/’ at the end to a slash-separated path
- allowing hostnames with the full set of legal characters instead of just a-z

### Lexing and Parsing

We're going to build some classes that read and interpret these languages.

Strings and streams have a fine-grained representation of the input (characters), but the kinds of input we want to process usually have bigger symbols than that. So it will be useful to divide input into two steps:

- **lexical analysis**, or **lexing**, which transforms the stream of characters into a stream of higher-level symbols, like words, or HTML tags, or whole chunks of text. These symbols are usually called **lexemes** or **tokens**.
- **parsing**, which takes the stream of tokens and interprets them. The parser is responsible for knowing the relationships between them (e.g., checking that each `<i>` is matched with a corresponding `</i>`).

In typical practice, these two steps are designed as independent state machines, a lexer and a parser, that interact through a clean interface: the output elements of the lexer are consumed by the parser. This is an instance of a general software design principle called **separation of concerns**: the lexer worries about lexing (e.g., what an individual HTML tag like “<i>” should look like), and the parser worries about parsing (e.g. that “<i>” should precede “</i>”). Although they are closely coupled in
the sense that you can’t use the parser without the lexer, they still have a clear contract between them.

**Lexical Analysis**

Lexical analysis takes a stream of fine-grained, low-level symbols (e.g., characters) and aggregates them into a sequence of higher-level symbols (e.g. words), called lexemes. The process is also called *tokenization*, and its output symbols are *tokens*.

Tokenization makes the second stage of input handling (parsing) simpler, by abstracting out some of the details of the input. For example, a lexer for Java throws away information that the compiler doesn’t care about, like whitespace and comments.

```java
/** square a number */
int square (int x) {
    return x*x;
}
```

might produce a token sequence like:

```java
int  square  (  int  x  )  {  return  x  *  x  ;  }
```

It can also combine symbols into classes that are useful to the parser. For Java, for example, user-defined names are typically grouped into a single kind of token, *identifier* or *id* for short, that also carries along information about the particular name:

```java
int   Id("square")   (    int    Id("x")    )    {
    return   Id("x")   *    Id("x")   ;    }
```

So tokens are not just strings, but might be objects with fields. In Java, an *enum* class is a useful way to define tokens.

Different languages call for different kinds of tokenization. In Python, for example, you wouldn’t throw away all whitespace entirely; you’d have tokens for newlines and tokens for indentation, since those affect Python statement structure. In natural language processing (like English), a tokenizer might detect parts of speech (nouns, adjectives) and undo morphology (e.g. “mice” becomes “mouse+plural”).

For our markup languages, we certainly want tokens for the italic syntax (*, <i>, \em, etc.). We might also considering throwing away whitespace, but let’s not; whitespace is actually significant in these formats. Markdown pays attention to blank lines, for example. So instead we’ll just treat all other text as a single kind of token called *text*, like this:

**HTML**

This is an *italic* word.

```html
Text("This is an ") <i> Text("italic") </i> Text(" word.")
```

**Markdown**

Words are _italic_.

```markdown
Text("Words are ") _ Text("italic") _ Text(".")
```
Lexer

A lexer is a state machine that does lexing. Like an iterator, a lexer typically has one method `next()` that returns the next token in the sequence. Inside the lexer is a state machine that processes the characters of the input stream in order to generate the token sequence.

You also have to make a design decision about how the lexer signals the end of the sequence. One option is the approach taken by Iterator: a method `hasNext()` that indicates whether another token is available. Another option is a special END token; InputStreams use this technique when `read()` returns -1. Another option is throwing an exception from `next()`. Some of these alternatives are discussed in the lecture on Specifications.

Grammars for Markdown and HTML

Here’s the grammar for our simplified version of markdown:

```
markdown ::= ( normal | italic ) *
italic ::= ‘`’ text ‘`’
normal ::= text
text ::= [^ _] *
```

Here’s the grammar for our simplified version of HTML, which allows italic regions to be nested inside other italic regions:

```
html ::= ( normal | italic ) *
Italic ::= ‘<i>’ html ‘</i>’
normal ::= text
text ::= [^ < >] *
```

Regular Grammars and Regular expressions

A regular grammar has a special property: by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only terminals and operators on the right-hand side. This “compiled” form of a regular grammar is called a regular expression.

Our URL grammar was regular. By replacing nonterminals with their productions, it can be reduced to a single expression:

```
url ::= ‘http://’ ([a-z]+ ‘.’)+ [a-z]+ (‘:’ [0-9]+) ‘/’
```

The markdown grammar is also regular:

```
markdown ::= ([^ _]* | ‘`’[^ _]* ‘`’)*
```

Regular expressions are also called regexes for short. A regex is far less readable than the original grammar, because it lacks the nonterminal names that documented the meaning of each subexpression. But a regex is fast to implement, and there are libraries in many programming languages that support regular expressions. More on this later in the lecture.

Using regular expressions in Java

Regular expressions (“regexes”) are even more widely used in programming tools than parser generators, and you should have them in your toolbox too.

In Java, you can use regexes for manipulating strings (see `String.split`, `String.match`, `java.util.regex.Pattern`). They’re built-in as a first-class feature of modern scripting languages like
Perl, Python, Ruby, and Javascript, and you can use them in many text editors for find and replace. Regular expressions are your friend! Most of the time. Here are some examples:

- replace all runs of whitespace with a single space, strip leading and trailing spaces:

  ```java
  string.replace("\s+", " ").replace("^\s+", "").replace("\s+$", "");
  ```

- match a URL:

  ```java
  Matcher m = Pattern.compile("http://([a-z]+\.|[a-z]+(:[0-9]+)?\/)").matcher(string);
  if (m.matches()) {
    // then string is a url
  }
  ```

- extract part of an HTML tag

  ```java
  Matcher m = Pattern.compile("<a href='([^\']*)'>").matcher(string);
  if (m.matches()) {
    m.group(1) is the desired URL
  }
  ```

**Context-Free Grammars**

In general, a language that can be expressed with our system of grammars is called **context-free**. Not all context-free languages are also regular; that is, some grammars can't be reduced to single nonrecursive productions.

Our HTML grammar is context-free but not regular.

The grammars for most programming languages are also context-free. In general, any language with nested structure (like nesting parentheses or braces) is context-free but not regular. That description applies to the Java grammar, shown here in part:

```plaintext
statement ::= block
  | ‘if’ ‘(‘ expression ‘)’ statement ‘(else’ statement)?
  | ‘for’ ‘(‘ forinit? ‘;’ expression? ‘;’ forupdate? ‘)’ statement
  | ‘while’ ‘(‘ expression ‘)’ statement
  | ‘do’ statement ‘while’ ‘(‘ expression ‘)’ ‘;’
  | ‘try’ Block ( catches | catches? ‘finally’ block )
  | ‘switch’ ‘(‘ expression ‘)’ ‘{‘ switchgroups ‘}’
  | ‘synchronized’ ‘(‘ expression ‘)’ block
  | ‘return’ expression? ‘;’
  | ‘throw’ expression ‘;’
  | ‘break’ identifier? ‘;’
  | ‘continue’ identifier? ‘;’
  | expression ‘;’
  | identifier ‘:’ statement
  | ‘;’
```

For a more precise treatment of the concepts of regular and context-free, see 6.045.
Parser generators

A *parser generator* is a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar. The generated source code typically includes both a lexer and a parser.

Antlr is a mature and widely-used parser generator for Java (and other languages as well). The remainder of this lecture will get you started with Antlr. If you run into trouble and need a deeper reference, you can look at:

- **The Definitive Antlr 4 Reference.** A book about Antlr, both tutorial and reference.
- **Antlr 4 Documentation Wiki.** Concise documentation of the grammar file syntax.
- **Antlr 4 Runtime API.** Reference documentation for Antlr's Java classes and interfaces.

The boldfaced keywords above will help you find these things on the web.

An Antlr Grammar

Here is what our HTML grammar looks like as an Antlr source file:

```antlr
grammar Html;

/*@  These are the lexical rules. They define the tokens used by the lexer.  */
START_ITALIC : '<i>' ;
END_ITALIC   : '</i>' ;
TEXT      : ~[<>]+ ;

/*@  These are the parser rules. They define the structures used by the parser.  */
root : html EOF ;
html : ( normal | italic ) * ;
normal : TEXT ;
italic : START_ITALIC html END_ITALIC ;

```

Let’s break it down.

- An Antlr rule consists of a token or nonterminal (like TEXT or root in the examples above), followed by a colon, followed by its definition, terminated by a semicolon.

The first group of rules are token definitions, so they describe the **lexer**:

```
START_ITALIC : '<i>' ;
END_ITALIC   : '</i>' ;

```

- START_ITALIC and END_ITALIC are tokens for the start and end tag. Literal strings are quoted with single quotes in Antlr. Capitalized identifiers, like START_ITALIC, are always tokens in Antlr.

```
TEXT      : ~[<>]+ ;
```

```
TEXT is a token representing all other text. In normal regular expression syntax that we used earlier in this lecture, we wrote this as \[~<\>\]+, to represent all characters except < and >, repeated at least once. ANTLR uses a slightly different syntax -- ~ means not, so ~\[<\>\] matches any character except < and >.

Tokens can be defined using regular expressions. For example, here are some other token definitions we used in the URL grammar earlier in the lecture, now written in ANTLR syntax and with ANTLR’s required naming convention:

```
IDENTIFIER : [a-z]+;
PORT : [0-9]+;
```

The second group of rules are the grammar rules, so they describe the parser:

```
root : html EOF ;
html : ( normal | italic ) * ;
normal : TEXT ;
italic : START_ITALIC html END_ITALIC ;
```

Nonterminals in ANTLR have to be lowercase.

- root is the entry point of the grammar. This is the nonterminal that the whole input needs to match. We don’t have to call it root, it can be an arbitrary nonterminal.
- EOF is a special token (defined by ANTLR) that means the end of the input (literally, “end of file”, though your input may also come from a string or a network connection rather than just a file.)

```
html : ( normal | italic ) * ;
```

This rule shows that ANTLR rules can have alternation |, grouping with parentheses, and repetition (*) and +) in the same way we’ve been using in the grammars earlier in this lecture. Optional parts can be marked with ?, just like we did earlier. There aren’t any rules with optional parts in this particular grammar.

More about ANTLR’s grammar file syntax can be found in Chapter 5 of the Definitive ANTLR 4 Reference.

Generating the lexer and parser

The ANTLR parser-generator tool converts a grammar source file like Html.g4 source file into Java classes that implement a lexer and parser. To do that, you need to go to a command prompt (Terminal or Command Prompt) and run a command like this:

```
java –jar antlr.jar Html.g4
```

You’ll need to make sure you’re in the right folder (where Html.g4 is) and that antlr.jar is in there too (or refer to it where it is – it may be a relative path like ../../../antlr.jar).

Assuming you don’t have any syntax errors in your grammar file, the parser generator will produce the following files in your folder:

- HtmlLexer.java is the lexer. It implements ANTLR’s Lexer interface. This class turns a stream of characters (like a string) into a stream of tokens (START_ITALIC, END_ITALIC, TEXT).
- HtmlParser.java is the parser, implementing the Parser interface. This class turns a stream of tokens into a parse tree.
- `HtmlListener.java` and `HtmlBaseListener.java` will let you write code that walks over Antlr's parse tree, as explained below.
- `Html.tokens` and `HtmlLexer.tokens` are text files that list the tokens that Antlr found in your source file. These aren't needed for a simple parser, but they're needed when you include grammars inside other grammars.

Make sure that you:

- **Never edit the files generated by Antlr.** The right way to change your lexer and parser is to edit the grammar source file, `Html.g4`, and then regenerate the Java classes.
- **Regenerate the files whenever you edit the grammar file.** This is easy to forget when Eclipse is compiling all your Java source files automatically. Eclipse does not regenerate your lexer and parser automatically. Make sure you rerun the `java -jar` command whenever you change your `.g4` file.

More about using the Antlr parser-generator to produce a lexer & parser can be found in Chapter 3 of the Definitive ANTLR 4 Reference.

### Calling the lexer and parser

Now that you've generated the Java classes for your lexer and parser, you'll want to use them from your own code.

First we need to make a stream of characters to feed to the lexer. Antlr has a class `ANTLRInputInputStream` that makes this easy; it can take a String, or a Reader, or an Input Stream. Here we are using a string:

```java
CharStream stream = new ANTLRInputInputStream("This is my <i>example of italics</i>\n");
```

Next, we create an instance of the lexer class that our grammar file generated, and pass it the character stream:

```java
HtmlLexer lexer = new HtmlLexer(stream);
TokenStream tokens = new CommonTokenStream(lexer);
```

The result is a stream of tokens, which we can then feed to the parser:

```java
HtmlParser parser = new HtmlParser(tokens);
```

To actually do the parsing, we call a particular nonterminal on the parser. The generated parser has one method for every nonterminal in our grammar, including `root()`, `html()`, `normal()`, `italic()`. We want to call the nonterminal that represents the set of strings that we want to match – in this case, `root()`. Calling it produces a parse tree:

```java
ParseTree tree = parser.root();
```

For debugging, we can then print this tree out:

```java
System.err.println(tree.toStringTree(parser));
```

Or we can display it in a handy graphical form:

```java
((RuleContext)tree).inspect(parser);
```

which displays the parse tree:
Constructing an abstract syntax tree

So we’ve used the lexer/parser to turn a stream of characters into a parse tree, showing how the grammar matches the stream. The final step is to do something with this parse tree. We’re going to translate it into a value of a recursive abstract datatype, like we talked about last lecture.

Recursive abstract datatypes are often used to represent an expression in a language, like HTML, or Markdown, or Java, or algebraic expressions. A recursive abstract datatype that represents a language expression is called an abstract syntax tree (AST).

Here’s the definition of the recursive abstract datatype that we’re going to use for documents with italics in them:

```
Document = Text(string:String)
  + Italic(child:Document)
  + Sequence(parts: ImList<Document>)
```

Document values will be able to represent documents written either in HTML or Markdown, or in fact RTF or Microsoft Word or any other format (limited to its text and italics, only). In that sense, the syntax is abstract – it has no details about the format.

By contrast, the parse tree that we just generated with our parser is a concrete syntax tree. It’s called concrete, rather than abstract, because it contains far more details about how the expression is represented in actual characters – that italics are bracketed in `&lt;i&gt;` and `&lt;/i&gt;`, for example. We’ve already seen two different concrete syntaxes for representing documents with italics – HTML and Markdown. An HTML document and a Markdown document may produce different concrete syntax trees, but they could map to the same abstract syntax tree.

Now let’s see how to translate from the parse tree into an instance of the Document type. To do this, we’ll use a ParseTreeWalker, which is an Anlr class that walks over a parse tree, visiting every node in order, top-to-bottom, left-to-right. As it visits each node in the tree, the walker will call methods on a TreeListener that we provide.

```java
ParseTreeWalker walker = new ParseTreeWalker();
HtmlListener_DocumentCreator listener = new HtmlListener_DocumentCreator();
walker.walk(listener, tree);
```

We have to provide a definition of the class HtmlTreeListener, which implements the HtmlListener interface that the parser generator produced from our grammar file. Just to warm up, here’s a simple implementation of HtmlTreeListener that just prints a message every time the walker calls us, so we can see how it gets used:

```java
private static class HtmlListener_PrintEverything extends HtmlBaseListener {
    public void enterRoot(HtmlParser.RootContext ctx) { System.err.println("entering root"); }
    public void exitRoot(HtmlParser.RootContext ctx) { System.err.println("exiting root"); }
}
```
public void enterNormal(HtmlParser.NormalContext ctx) { System.err.println("entering normal"); }
public void exitNormal(HtmlParser.NormalContext ctx) { System.err.println("exiting normal"); }
public void enterHtml(HtmlParser.HtmlContext ctx) { System.err.println("entering html"); }
public void exitHtml(HtmlParser.HtmlContext ctx) { System.err.println("exiting html"); }
public void enterItalic(HtmlParser.ItalicContext ctx) { System.err.println("entering italic"); }
public void exitItalic(HtmlParser.ItalicContext ctx) { System.err.println("exiting italic"); }

If we walk over the parse tree with this listener, then we’ll see the following output:

entering root
entering html
entering normal
exiting normal
entering italic
entering html
entering normal
exiting normal
exiting html
exiting italic
exiting html
exiting root

Now we’ll create a listener that constructs a Document tree while it’s walking over the parse tree. Each parse tree node will correspond to a Document object:

- italic nodes will create Italic objects
- normal nodes will create Text objects
- html nodes will create Sequence objects

Whenever the walker exits each node of the parse tree, we’ll have walked over the entire subtree under that node, so we’ll create the next Document object at exit time. But we have to keep track of all the children that were created during the walk over that subtree. We’ll use a stack to store them.

Here’s the code:

private static class HtmlListener_DocumentCreator extends HtmlBaseListener {
    private Stack<Document> stack = new Stack<Document>();

    public void exitNormal(NormalContext ctx) {
        TerminalNode token = ctx.TEXT();
        String text = token.getText();
        Document node = new Text(text);
        stack.push(node);
    }
}
```java
public void exitItalic(ItalicContext ctx) {
    Document child = stack.pop();
    Document node = new Italic(child);
    stack.push(node);
}

public void exitHtml(HtmlContext ctx) {
    ImList<Document> list = new Im.Empty<Document>();
    int numChildren = ctx.getChildCount();
    for (int i = 0; i < numChildren; ++i) {
        list = list.cons(stack.pop());
    }
    Document node = new Sequence(list);
    stack.push(node);
}

public void exitRoot(RootContext ctx) {
    // do nothing, because the top of the stack should have the node already in it
    assert stack.size() == 1;
}

public Document getDocument() {
    return stack.get(0);
}
}
```

More about Antlr’s parse-tree listeners can be found in Section 7.2 of the Definitive ANTLR 4 Reference.

**Handling errors**

By default, Antlr lexers and parsers print errors to the console. In order to make the lexer/parser modular, however, we’ll need to handle those errors differently. You can attach an ErrorListener to a lexer or a parser which throws an exception instead when it encounters an error during lexing or parsing. The Html.g4 file defines a method reportErrorsAsExceptions() which does this. So if you copy the technique used in this grammar file, you can call:

```java
lexer.reportErrorsAsExceptions();
pARSER.reportErrorsAsExceptions();
```

right after you create your lexer and parser, respectively. Then when you call parser.root(), it will throw an exception as soon as the lexer encounters something it can’t turn into a token, or the parser encounters a sequence of tokens that it can’t match.

This is a relatively simplistic approach to handling errors. Antlr offers more sophisticated forms of error recovery as well. To learn more, see Chapter 9 in the Definitive Antlr 4 Reference.

**Summary**

Machine-processed textual languages are ubiquitous in computer science. Context-free grammars are the most popular formalism for describing such languages, and regular languages are an important subclass of grammars, expressible without recursion. The classic way of implementing a language processor is to split it into a lexer and a parser.