

L5: Regular Expressions & Grammars

Today

- Lexing and parsing
- Grammars
- Regular expressions

Required reading (from the Java Tutorial)

- [Enums](#)
- [Regular Expressions](#)

Markup

For today's working examples, we'll be using several 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_`.

LaTeX

In LaTeX, `{\em italics}` are used to show `{\em emphasis}`, unless you're nesting emphasized text inside other emphasized text}.

State machine review

Let's start by drawing state machines representing the behavior of a renderer for these markup languages. All three have two states:

Normal Italic

What differs are the transitions between them.

The state machines alone are dissatisfying – they don't tell the whole story about the language. In particular, they don't show the *structure* of the language, particularly of LaTeX.

Reading Input

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

The first thing we need to do is decide on the set of events we want our state machine to use. Strings and streams give us very fine-grained events, like 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 `<i>` precedes `</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 events 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.

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

might produce a token sequence like:

```
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:

```
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 (Latex and markdown pay 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 <i>italic</i> word.

Text(“This is an ”) <i> Text(“italic”) </i> Text(“ word.”)
```

Markdown

```
Words are _italic_.
```

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

LaTeX

```
You can {\em nest {\em italics} in Latex}
```

```
Text("You can ") {\em Text(" nest ") {\em Text("italics") }  
Text(" in Latex") }
```

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.

Grammar

- a grammar defines a set of sentences
- a sentence is a sequence of symbols (tokens, also called *terminals*)
- a grammar is a set of productions
- each production defines a non-terminal
- a non-terminal is a variable that stands for a set of sentences

By convention, nonterminals are capitalized, and terminals are lowercase.

production has form

- non-terminal ::= expression of terminals and non-terminals and operators

The three operators are:

- sequence: $A ::= B C$ an A is a B followed by a C
- iteration: $A ::= B^*$ an A is zero or more B's
- choice: $A ::= B \mid C$ an A is a B or a C

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

- option: $A ::= B?$ an A is a B or is empty
- grouping: $A ::= (B\ C)^*$ parentheses for grouping an A is zero or more B-C pairs
- 1+iteration: $A ::= B^+$ is equivalent to $A ::= BB^*$ an A is one or more B's
- character classes: $A ::= [abc]$ is equivalent to $A ::= a \mid b \mid c$
 $A ::= [\wedge b]$ is equivalent to $A ::= a \mid c \mid d \mid e \mid f \mid \dots$ (all other characters)

example:

grammar

$URL ::= \text{Protocol} \text{ :// } \text{Address}$

$\text{Address} ::= \text{Domain} \text{ . } \text{TLD}$

$\text{Protocol} ::= \text{http} \mid \text{ftp}$

$\text{Domain} ::= \text{mit} \mid \text{apple} \mid \text{pbs}$

$\text{TLD} ::= \text{com} \mid \text{edu} \mid \text{org}$

terminals are

$\text{://, ., http, ftp, mit, apple, pbs, com, edu, org}$

non-terminals are

$\text{TLD} = \{ \text{com, edu, org} \}$

$\text{Domain} = \{ \text{mit, apple, pbs} \}$

$\text{Protocol} = \{ \text{http, ftp} \}$

$\text{Address} = \{ \text{mit.com, mit.edu, mit.org, apple.com, apple.edu, apple.org, pbs.com, pbs.edu, pbs.org} \}$

$\text{URL} = \{ \text{http://mit.com, http://mit.edu, ..., ftp://mit.com, ...} \}$

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

$\text{Markdown} ::= (\text{Normal} \mid \text{Italic})^*$

$\text{Italic} ::= _ \text{Text} _$

$\text{Normal} ::= \text{Text}$

$\text{Text} ::= [\wedge _]^*$

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

$\text{Html} ::= (\text{Normal} \mid \text{Italic})^*$


$\text{Italic} ::= <i> \text{Html} </i>$

$\text{Normal} ::= \text{Text}$

$\text{Text} ::= [\wedge_]^*$

And here is our Latex grammar:


$\text{Latex} ::= (\text{Normal} \mid \text{Italic})^*$

$\text{Italic} ::= \{\backslash\text{em Latex}\}$ 

$\text{Normal} ::= \text{Text}$

$\text{Text} ::= [\wedge_]^*$

Regular Grammars

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*. 

The markdown grammar is regular. By replacing nonterminals with their productions, it can be reduced to a single nonrecursive production:

$\text{Markdown} ::= ([\wedge_]^* \mid _ [\wedge_]^* _)^*$

The expression on the righthand side, consisting only of terminals and operators, is called a **regular expression**. It’s far less readable than the grammar, but it’s fast to implement, and there are many libraries in many programming languages that support regular expressions (called *regexes* for short). More on this later in the lecture.

A grammar that **can’t be reduced to a single nonrecursive production is called *context-free***. Both the HTML and Latex grammars are context-free. 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. Here’s part of the grammar for Java statements:

$\text{Statement} ::= \text{Block}$

| if ParExpression Statement [else Statement]
| for (ForInit? ; Expression? ; ForUpdate?) Statement
| while (Expression) Statement
| do Statement while (Expression) ;
| try Block (Catches | Catches? finally Block)
| switch (Expression) { SwitchBlockStatementGroups }
| synchronized ParExpression Block
| return Expression? ;
| throw Expression ;
| break Identifier? ;
| continue Identifier? ;
| ExpressionStatement
| Identifier : Statement
| ;

Grammars and State Machines

regular grammars vs state machines

▸ a state machine's trace set is prefix closed: if t^e is a trace, so is t ▸ regular grammars can express trace sets that are not prefix closed

traces of $(\text{up down})^*$ include $\langle \text{up}, \text{down} \rangle$ but not $\langle \text{up} \rangle$

▸ so grammars are more expressive

but can add "final" states to state diagrams

▸ then define (full) traces as those that go from initial to final states

▸ now regular grammars and machines are equally expressive

▸ they both define regular languages

in practice

▸ use state machines for non-terminating systems

▸ use grammars for terminating and non-terminating systems

Recursive descent parsing and evaluation

The grammar guides the design of your parser class. The code below shows an example of a recursive-descent parser for the markdown grammar.

```
/**
 * A Gallileo object is a parser/evaluator for markdown that scrambles
 * italic text (generates a random anagram of each italic part) so that Kepler
 * can't read it.
 */
public class Gallileo {

    private final MarkdownLexer lex;

    public Gallileo(String markdown) {
        this.lex = new MarkdownLexer(markdown);
    }

    /**
     * Evaluate the input text, scrambling italic sections.
     * Can be called only once on a given object.
     * Modifies this object, consuming all the text.
     * @return string of text with markdown formatting removed
     * and italic sections replaced by a random anagram.
     * For example, new Gallileo("The killer was _Mrs. White_").eval()
     * ==> "The killer was hrW.sM tie"
     */
    public String eval() {
        return evalMarkdown();
    }

    // Grammar:
    // Markdown ::= (Normal | Italic)*
    // Normal ::= Text
    // Italic ::= _ Text? _
```

```

//
// (Text and _ are tokens generated by MarkdownLexer)

/**
 * Evaluates the Markdown production of the grammar.
 * Modifies lex by consuming all the remaining tokens.
 * @return evaluated string
 */
private String evalMarkdown() {
    StringBuilder sb = new StringBuilder();

    for (Token tok = lex.next(); tok.getType() != Type.EOF; tok =
lex.next()) {
        switch (tok.getType()) {
            case UNDERLINE:
                sb.append(evalItalic(tok));
                break;
            case TEXT:
                sb.append(evalNormal(tok));
                break;
            default:
                throw new AssertionError("unexpected token: " + tok.getType());
        }
    }

    return sb.toString();
}

/**
 * Evaluates the Normal production of the grammar.
 * Modifies lex by consuming an entire production, including the last token
of the production.
 * @param tok Token that started this production (required to be TEXT)
 * @return evaluated string
 */
private String evalNormal(Token tok) {
    // normal text isn't changed by this process, just return it as-is
    return tok.getValue();
}

/**
 * Evaluates the Italic production of the grammar.
 * Modifies lex by consuming an entire Italic production, including its
final token.
 * @param tok Token that started this production (required to be UNDERLINE)
 * @return evaluated string
 */
private String evalItalic(Token tok) {
    StringBuilder sb = new StringBuilder();

    // the passed in tok is UNDERLINE; skip it and advance to the next

    // note that this code actually evaluates _ TEXT* _, not just _ TEXT? _
    for (tok = lex.next(); tok.getType() != Type.EOF && tok.getType() !=
Type.UNDERLINE; tok = lex.next()) {
        if (tok.getType() == Type.TEXT) {
            // collect and shuffle the text
            sb.append(shuffle(tok.getValue()));
        } else {
            throw new AssertionError("unexpected token: " + tok.getType());
        }
    }
}

```

```

        return sb.toString();
    }

    /**
     * Make a random anagram of a string.
     * @param s string to rearrange
     * @return a random permutation of the characters in s.
     * For example, shuffle("abc") might return "bca" or "cba" or "abc".
     */
    static String shuffle(String s) {
        // this is not the best way to implement this -- what would be better?

        // split with empty-string separator to get each char as a string
        String[] a = s.split(""); // e.g. "", "a", "b", "c" (produces an
        // extra empty string, but that won't hurt)

        List<String> l = Arrays.asList(a);
        Collections.shuffle(l); // now it's shuffled, e.g. "a", "", "c", "b"

        // glue the shuffled list back together into one string
        StringBuilder sb = new StringBuilder();
        for (String t : l) {
            sb.append(t);
        }

        return sb.toString();
    }

    /**
     * Main method.
     */
    public static void main(String[] args) {
        Gallileo g = new Gallileo("I've discovered that _Saturn has ears_.
        Suck it, Kepler!");
        String message = g.eval();
        System.out.println(message);
    }
}

```

Parser generators

For some grammars, particularly more complex context-free grammars, you need heavier machinery. *Parser generators* are a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates parser code for that grammar – typically both a lexer and a parser. JavaCC is a mature and widely-used parser generator for Java.

Using regular expressions

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:


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

extract part of an HTML tag

```
Matcher m = Pattern.compile("<a href='([^\']*)'>").matcher(string);
```

```
if (m.matches()) {
```

```
    m.group(1) is the desired URL
```

```
}
```

Risks of regular expressions

safe from bugs?

easy to understand?

ready for change?

MIT OpenCourseWare
<http://ocw.mit.edu>

6.0001: Introduction to Computer Science and Programming
Fall 2011

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.