Some Applications of Regular Languages

- Automata = finite state machines (or extensions thereof) used in many disciplines
- Efficient string searching
- Pattern matching with regular expressions (example: Unix grep utility)
- Lexical analysis (a.k.a. scanning, tokenizing) in a compiler (the topic of a lecture later in the course)
Markov Models

DFAs in User Interfaces

Example:
Black Diamond Storm headlamp provides access to all features via a single button. Can construct a DFA to explain the interface.

Naïve String Searching

How to search for `abbaba` in `abbabcabbabbaba`?

```
  a   b   b   a   b   c   a   b   b   a   b   b   a   a
  a   b   b   a   b   a   b   a   b   b   a   b   b   a
  a   b   c   b   a   b   a   b   c   b   c   b   c   b
  a   b   b   c   a   b   b   c   a   b   b   a   b   a
```

More Efficient String Searching

Knuth-Morris-Pratt algorithm: construct a DFA for searched-for string, and use it to do searching.

```
  b, c   a   a   b   a
  b   b   b   a   b   a
  a   b   b   a   b   a
```

How to construct this DFA automatically?
Pattern Matching with Regular Expressions

Can turn any regular expression (possibly extended with complement, intersection, and difference) into a DFA and use it for string searching.

This idea is used in many systems/languages:
- **grep**: Unix utility that searches for lines in files matching a pattern. ("grep" comes from g/re/p command in the ed editor.)
- **sed**: Unix stream editor
- **awk**: text-manipulation language
- **Perl**: general-purpose programming language with built-in pattern matching
- **JavaScript**: can use regular expressions for form validation.
- **Java, Python, etc.**: have support for regular expressions.
- **Emacs**: supports regular expression search

<table>
<thead>
<tr>
<th>Some grep Patterns</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>the character 'c'</td>
</tr>
<tr>
<td>.</td>
<td>any character except newline</td>
</tr>
<tr>
<td>[a-zA-Z0-9]</td>
<td>any alphanumeric character</td>
</tr>
<tr>
<td>['d-g]</td>
<td>any character except lowercase d,e,f,g</td>
</tr>
<tr>
<td>\w</td>
<td>synonym for [a-zA-Z0-9]</td>
</tr>
<tr>
<td>[[:space:]]</td>
<td>all whitespace characters</td>
</tr>
<tr>
<td>^</td>
<td>beginning of line</td>
</tr>
<tr>
<td>$</td>
<td>end of line</td>
</tr>
<tr>
<td>\s</td>
<td>end of word</td>
</tr>
<tr>
<td>r1?r2</td>
<td>r1 followed by r2, where r1, r2 are reg. exps.</td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
</tr>
<tr>
<td>r*</td>
<td>zero or more rs, where r a reg. exp.</td>
</tr>
<tr>
<td>r+</td>
<td>one or more rs</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one rs</td>
</tr>
<tr>
<td>r{n}</td>
<td>exactly n rs</td>
</tr>
<tr>
<td>r{n,m}</td>
<td>n or more rs</td>
</tr>
<tr>
<td>r(n,m)</td>
<td>between n and m rs</td>
</tr>
<tr>
<td>(r)</td>
<td>r (paren for grouping)</td>
</tr>
<tr>
<td>\n</td>
<td>the substring previously matched by the nth parenthesized subexpression of the regular expression</td>
</tr>
</tbody>
</table>

Some grep Examples

As a rule, grep patterns should be double-quoted to prevent Linux from interpreting certain characters specially. (But \ is still a problem, as we'll soon see.)

cd ~cs235/public_html
grep "a.*b.*c.*d" words.txt

grep "^a.*b.*c.*d" words.txt

grep "a.*b.*c.*d$" words.txt

grep "^a.*b.*c.*d$" words.txt

grep "^.*a.b.*c.*d$" wordlists/*words* (in Scowl final database)

cd ~cs230/archive/cs230_fall04/download/collections

grep "delete[[:space:]]"*(Object" *_.java

grep "/.*sorted" *_.java

A Powerful Combination: find With grep

Unix's **find** command enumerates all files in a directory. E.g

cd ~cs230/archive/cs230_fall04/download/

```
find .
```

In combination with **grep**, it can search all these files!

```
find . | xargs grep "delete[[:space:]]"*(Object"
```

```
fine -exec grep -H "delete[[:space:]]"*(Object" {}
```
Escapes in Grep Patterns

grep patterns use special metacharacters that (at least in some contexts) do not stand for themselves:

`? + | ( ) { } . ^ $ \ [ ]`

In order to reference the blue characters as themselves, it is necessary to escape them with a backslash. E.g.,
- `$` is a pattern that matches the end of line
- `\$` is a pattern that matches the dollar sign character
- `\` is a pattern that matches the backslash character
- `\\` is a pattern that matches two backslash characters in a row

But the backslash character is also an escape character in Linux! To safely pass backslashes from Linux to grep, you should* type two backslashes for every backslash you wish to send to grep. E.g.

- `\$` searches for the dollar sign character
- `\\` searches for a single backslash
- `\\\` searches for two backslash characters in a row

*In some, but not all cases, a single backslash will suffice.

What About the Red Metacharacters?

The red metacharacters are handled in a rather confusing way:

`? + | ( ) { }`

In the basic regular expressions used by grep, these characters stand for themselves and must be escaped to have the metacharacter meaning. E.g.
- `grep "(ab)*"` searches for the substring "(ab)*"
- `grep "(ab){2}"` searches for the substring "(ab){2}"
- `grep "\(ab\)\{2\}"` searches for any nonempty sequence of `ab`
- `grep "\(ab\)\{2\}"` searches two `ab` in a row
- `grep "\(\)\{2\}"` searches for two consecutive occurrences of the same character

In the extended regular expressions used by `grep -E` and `egrep`, these characters are metacharacters and must be escaped to stand for themselves.
- `egrep "(ab)*"` searches for any nonempty sequence of `ab`
- `egrep "(ab){2}"` searches two `ab` in a row
- `egrep "\(ab\)\{2\}"` searches for the substring "(ab){2}"
- `egrep "\(\)\{2\}"` searches for two consecutive occurrences of the same character

Moral of the story: use `egrep` instead of `grep!

More Practical Examples

1. Write an egrep regular expression that matches only well-formed short FirstClass usernames (e.g., fturbak, gdome, etc.)

Such usernames consist of at least 2 and at most 8 characters and are sequences of lowercase letters followed by at most 2 digits.

2. Write an egrep regular expression that matches only well-formed email address of the form `username@server.domain`, where

   - `username` is any sequence of letters, numbers, underscores, and dots that begins with a letter;
   - `Server` is any sequence of letters and numbers that begins with a letter;
   - `Domain` is one of `com`, `edu`, or `gov`.

egrep Examples

cd ~cs235/public_html /wordlists
egrep "(ab){2}" *words*
egrep "(a.*b){2}" *words*
egrep "(a.*b.*){2}" *words*
egrep "(a.*b){1}" *words*
egrep "(a.*b){0}" *words*
egrep "(a.*b){1}" *words*
egrep "(a.*b){0}" *words*
egrep "(....){1}" *words*
egrep "(....){0}" *words*
egrep "(.).*(.)*\d{2}\*\1" *words*
egrep "((.)(.)*\d{2}\*\1" *words*
Regexp Support in Programming Languages

Many popular programming languages (Java, JavaScript, Python, Perl, etc.) have built-in or library support for regular expressions.

E.g. Dive Into Python chapter on regular expressions:
http://diveintopython.nfshost.com/regular_expressions

Javascript example (Tanner’10 photo upload site):

```javascript
function validRegistration() {
    var emailPattern =/^[a-zA-Z]{2,8}$/|^[a-zA-Z]{2,7}[0-9]$|^[a-zA-Z]{2,6}[0-9]{2}$/;
    var emailAddress = document.registrationForm.email.value;
    ...
    if (emailAddress.search(emailPattern) == -1) {
        document.getElementById("registration_status").innerHTML = "<span style='color:red;'>You must use a legal Wellesley email address.</span>";
        return false;
    } ...
}
```

Jamie Zawinski’s warning:

Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.
(quoted at end of Sec. 7.7, Dive Into Python)

Applications of Search/Pattern Matching

- Document/file search
- Antivirus software
  - many viruses have a characteristic signature = sequence of bytes
  - virus-writers can create polymorphic viruses that thwart signature-based attacks.
- DNA/protein analysis
  - DNA is a 4-character alphabet; proteins a 20-character alphabet
  - in practice, don’t look for exact matches but want “close” ones;
    this uses dynamic programming technology (see CS231).

Compiler Structure

Front End Example

```c
if (num > 0 && num <= top) {
    return c*num
} else (return 0;)
```

```
if (num > 0 && num <= top) {
    return c*num
} else (return 0;)
```

Lexer (ignores whitespace, comments)

Parser (creates AST)
Lexical Analysis

Lexical analysis = breaking programs into tokens, the first stage of a compiler.

The structure of tokens can be specified by regular expressions.

Example: the ML-Lex tool can automatically derive a lexical analyzer from a .lex file --- a description of tokens specified by regular expressions.

We will spend an entire lecture on lexing later this semester.

Slip.lex Definitions and Rules

```
alpha=\[a-zA-Z\];
alphaNumUnd=\[a-zA-Z0-9_\];
digit=\[0-9\];
whitespace=\[\s\t\n\];
any=\[\^\];
%%
"print" => (PRINT);
{alpha}(alphaNumUnd)* => (ID(yytext));
{digit}+ => (INT(pluck(Int.fromString(yytext))));
"+" => (OP(Add));
"-" => (OP(Sub));
"*" => (OP(Mul));
"/" => (OP(Div));
"(" => (LPAREN); 
")" => (RPAREN);
"," => (COMMA);
";" => (SEMI);
"=" => (GETS);
{whitespace} => (lex());
{any} => ((* Signal a failure exception when encounter unexpected character.
 A more flexible implementation might raise a more refined
 exception that could be handled. *))
raise Fail("Slip scanner: unexpected character \""^ yytext ^ "\")
```

String matched by regular expression

Definitions

Discard current token and continue lexing

Rules

Remove SOME from option type.