COS 333: Advanced Programming Techniques

- Brian Kernighan
  - bwk@cs, www.cs.princeton.edu/~bwk
  - 311 CS Building
  - 609-258-2089 (but email is always better)

- TA's:
  - Junwen Lai, lai@cs, CS 217, 258-0451
  - Yong Wang, yongwang@cs, CS 103C, 258-0419

Today
- course overview
- administrative stuff
- regular expressions and grep

Check out the course web page
- notes, readings and assignments will be posted there
- Assignment 1 is posted

Do the survey if you haven’t already

Themes

- languages
  - C, C++, Java, AWK, Perl, Visual Basic, ...
  - programmable tools, application-specific languages

- tools for programming and programmers
  - how to use them (vocational training)
  - where did they come from and why
    how they have evolved, mutated, decayed
  - how they work inside
  - how to build your own

- building programs
  - design, interfaces
  - reuse, theft, prototyping, components
  - programs that write programs
  - portability, standards, style
  - debugging, testing
  - performance assessment and improvement
  - tricks of the trade
  - tradeoffs, compromises, engineering versus science

- history and culture of programming
(Very) Tentative Outline

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feb 3</td>
<td>regular expressions; grep</td>
</tr>
<tr>
<td>Feb 10</td>
<td>scripting languages: Awk &amp; Perl</td>
</tr>
<tr>
<td>Feb 17</td>
<td>more scripting: Perl, PHP(?)?, CGI</td>
</tr>
<tr>
<td>Feb 24</td>
<td>Java; object-oriented programming</td>
</tr>
<tr>
<td>Mar 2</td>
<td>Java; networking; databases; project</td>
</tr>
<tr>
<td>Mar 9</td>
<td>user interfaces, Swing</td>
</tr>
<tr>
<td>Mar 15</td>
<td>(spring break)</td>
</tr>
<tr>
<td>Mar 23</td>
<td>C++</td>
</tr>
<tr>
<td>Mar 30</td>
<td>C++, Standard Template Library</td>
</tr>
<tr>
<td>Apr 6</td>
<td>Visual Basic; COM, components</td>
</tr>
<tr>
<td>Apr 13</td>
<td>XML, web services; .net, C#</td>
</tr>
<tr>
<td>Apr 20</td>
<td>Tcl/Tk?, language tools?</td>
</tr>
<tr>
<td>Apr 27</td>
<td>?</td>
</tr>
<tr>
<td>May 5-6</td>
<td>project presentations</td>
</tr>
</tbody>
</table>

Some Mechanics

- **prerequisites**
  - C, Unix (COS 217)

- **5 programming assignments in first half**
  - posted on course web page
  - deadlines matter

- **group project in second half**
  - groups of 3-4: start identifying potential teammates
  - details in a few weeks
  - deadlines matter

- **monitor the web page**
  - readings for most weeks
  - notes generally posted ahead of time

- **class attendance and participation**
  - no midterm or final
  - sporadic unannounced short quizzes are possible
Regular expressions and grep

- **regular expressions**
  - notation
  - mechanization
  - pervasive in Unix tools
  - not in most general-purpose languages
    - though common in scripting languages and (some) editors
  - basic implementation is remarkably simple
  - efficient implementation requires theory and practice

- **grep is the prototypical tool**
  - people used to write programs for searching
    - (or did it by hand)
  - tools became important
  - tools are not as much in fashion today

### Grep regular expressions

- **c** any character matches itself, except for metacharacters . [ ] ^ $ * \\r
- **r₁,r₂** matches r₁ followed by r₂
- **.** matches any single character
- **[ ]** matches one of the characters in set ... a set like a-z or 0-9 includes any character in the range
- **[^ ]** matches one of the characters not in set a set like a-z or 0-9 includes any char in the range
- **^** matches beginning of line when ^ begins pattern
  - no special meaning elsewhere in pattern
- **$** matches end of line when $ ends pattern
  - no special meaning elsewhere in pattern
- **\** any regular expression followed by \ matches zero or more instances
- **\c** matches c unless c is ( ) or digit
- **\(...\)** tagged regular expression that matches ... the matched strings are available as \1, \2, etc.
Examples of matching

- `thing`: thing anywhere in string
- `^thing`: thing at beginning of string
- `thing$`: thing at end of string
- `^thing$`: string that contains only `thing`
- `^$`: empty string
- `.`: non-empty, i.e., at least 1 char
- `^`: matches any string, even empty
- `thing.$`: thing plus any char at end of string
- `thing\$`: thing, at end of string
- `\thing\`: `\thing` anywhere in string
- `[tT]thing`: thing or `Thing` anywhere in string
- `thing[0-9]`: thing followed by one digit
- `thing[^0-9]`: thing followed by a non-digit
- `thing[0-9][^0-9]`: thing followed by digit, then non-digit
- `thing1.*thing2`: thing1 then any text then thing2
- `^thing1.*thing2$`: thing1 at beginning and thing2 at end

egrep: fancier regular expressions

- `r*`: one or more occurrences of `r`
- `r?`: zero or one occurrences of `r`
- `r1|r2`: `r1` or `r2`
- `(r)`: `r` (grouping)

```
(\d+\.?\d+|\.\d+)([Ee][-+]?\d+)?
```
Grammar for egrep regular expresions

\[ r: c . ^ $ [ccc] [^ccc] \]
\[ r^* r^+ r^? \]
\[ r_1 r_2 \]
\[ r_1|r_2 \]
\[ (r) \]

Precedence:
* * + ? are higher than concatenation
which is higher than |


The grep family

* grep
  - basic matching
* egrep
  - fancier regular expressions
  - trades compile time and space for run time
* fgrep
  - parallel search for many fixed strings
* agrep
  - "approximate" grep: search with errors permitted

relatives that use similar regular expressions
- ed          original unix editor
- sed         stream editor
- vi, emacs, sam, ...  editors
- lex         lexical analyzer generator
- awk, perl, tcl, python, ...  scripting languages
- Java, CM ... libraries in mainstream languages

simpler variants
- filename "wild cards" in Unix and other shells
- "LIKE" operator in Visual Basic, SQL, etc.
Basic grep algorithm

while (get a line)
    if match(regexpr, line)
        print line

• (perhaps) compile regexpr into an internal
  representation suitable for efficient matching
• match() slides the regexpr along the input line,
  looking for a match at each point

Grep (TPOP, p. 226)

/* grep: search for regexp in file */
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];

    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
Match anywhere on a line

- look for match at each position of text in turn

```c
/* match: search for regexp anywhere in text */
int match(char *regexp, char **text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
```

Match starting at current position

```c
/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp+1, text+1);
    return 0;
}
```

- follow the easy case first: no metacharacters
- note that this is recursive
  - maximum depth: one level for each regexp character that matches
Matching * (repetitions)

- matchstar() called to match c*...
- matches if rest of regexp matches rest of input
  - null matches require test at the bottom

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
  do {    /* a * matches zero or more instances */
    if (matchhere(regexp, text))
      return 1;
  } while (*text != '\0' && (*text++ == c || c == '.'));
  return 0;
}

- finds the leftmost shortest match
  - just right for pattern matching in grep
  - NOT usually what we want in a text editor
  - null matches are surprising and rarely desired

Profiling: where does the time go

- count number of times each line is executed
  - measure how long each function takes
  - plus lots of other information

$ lcc -p grep.c
$ a.out x ../bib >foo
$ prof

<table>
<thead>
<tr>
<th>Time</th>
<th>Seconds</th>
<th>Cumsecs</th>
<th>Calls</th>
<th>msec/call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>52.1</td>
<td>0.37</td>
<td>0.37</td>
<td>4360969</td>
<td>0.0001</td>
<td>matchhere</td>
</tr>
<tr>
<td>16.9</td>
<td>0.12</td>
<td>0.49</td>
<td>4528173</td>
<td>0.0000</td>
<td>_mcount</td>
</tr>
<tr>
<td>11.3</td>
<td>0.08</td>
<td>0.57</td>
<td>31102</td>
<td>0.0026</td>
<td>match</td>
</tr>
<tr>
<td>8.5</td>
<td>0.06</td>
<td>0.63</td>
<td>946</td>
<td>0.11</td>
<td>_read</td>
</tr>
<tr>
<td>7.0</td>
<td>0.05</td>
<td>0.68</td>
<td>31642</td>
<td>0.0016</td>
<td>__memcpy</td>
</tr>
<tr>
<td>1.4</td>
<td>0.01</td>
<td>0.69</td>
<td>946</td>
<td>0.02</td>
<td>__filbuf</td>
</tr>
<tr>
<td>1.4</td>
<td>0.01</td>
<td>0.70</td>
<td>31103</td>
<td>0.0003</td>
<td>fprintf</td>
</tr>
<tr>
<td>1.4</td>
<td>0.01</td>
<td>0.71</td>
<td>32429</td>
<td>0.0003</td>
<td>strlen</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.71</td>
<td>1</td>
<td>0.00</td>
<td>main</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.71</td>
<td>1</td>
<td>0.00</td>
<td>grep</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.71</td>
<td>1</td>
<td>0.00</td>
<td>setprogn</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.71</td>
<td>1326</td>
<td>0.0005</td>
<td>printf</td>
</tr>
</tbody>
</table>

$ wc ../bib
31102  851820  4460056 ../bib
$ a.out x ../bib | wc
1326 39957 207477

- _mcount is profiling overhead
- note consistent counts
Statement frequency counts

```c
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {   /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (*regexp[0] == '$' && *regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}
```

- note conservation laws

How to make grep faster

- use optimization  (cc -O)
- change compilers  (lcc, gcc, vc++)
- code tuning
  - e.g., match calls matchhere many times
  - even though most of them must necessarily fail
  - because the target string doesn’t contain the first character of the pattern
- algorithm changes
Code tuning variant

- checks whether target contains first character of pattern before calling matchhere
  - unless it is x*

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
  char *p;
  if (regexp[0] == '^')
    return matchhere(regexp+1, text);
  if (regexp[0] != '\' && regexp[1] != '*')
    if ((p=strchr(text, regexp[0])) == NULL)
      return 0;
  do {
    /* must look even if string is empty */
    if (matchhere(regexp, p))
      return 1;
  } while (*p++ != '\0');
  return 0;
}

• is this faster?

Statement frequencies after change

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{ 31102:
  char *p;
  if (regexp[0] == '"')
    return 0; matchhere(regexp+1, text);
  if (regexp[0] != '.' && regexp[1] != '*')
    if ((p=strchr(text, regexp[0])) == NULL)
      return 29776:0;
  do { 1326:
    /* must look even if string is empty */
    if (1326:matchhere(regexp, p))
      return 1326:1;
  } while (1326:*p++ != '"');
  return 0:0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{ 2652:
  if (regexp[0] == '\0')
    return 1326:1;
  if (regexp[1] == '"')
    return 0:matchstar(regexp[0], regexp+2, text);
  if (regexp[0] == '$' && regexp[1] == '\0')
    return 0:*text == '\0';
  if ((regexp[0] == '\' && regexp[1] == '"' ||
    regexp[0] == '"' && regexp[1] == '\0')
    return 1326:matchhere(regexp+1, text+1);
  return 0:0;
}
Simple grep algorithm

- **best for short simple patterns**
  - e.g., `grep foo * [ch]`
  - most use is like this
  - reflects use in text editor for a small machine

- **limitations**
  - tries the pattern at each possible starting point
    - e.g., look for `aaaaab` in `aaaa...aaaaab`
    - potentially $O(mn)$ for pattern of length $m$
  - complicated patterns `(.*) .* (` require backup
    - potentially exponential
  - can’t do some things, like alternation (OR)

- **this leads to extensions and new algorithms**
  - `egrep` complicated patterns, alternation
  - `fgrep` lots of simple patterns in parallel
  - `boyer-moore` long simple patterns
  - `agrep` approximate matches

Finite state machines/finite automata

- **finite state machine**
  - a set of states
  - an alphabet (e.g., ascii)
  - transition rules: current state & input char $\rightarrow$ new state
  - a start state
  - a set of final "accepting" states

- **regular expressions are equivalent to finite state machines**
  - can go from one to the other mechanically

- `ab*c`

  ![Finite state machine diagram](image)

- `a*b`, if $n < 4`
  - can’t count: can’t handle arbitrary $n$ in a fixed number of states
  - can’t do palindromes: no memory
Non-deterministic finite automata (NDFA)

RE: "ab.*abab"

FSM: 0 1 2 3 4 5 6

input: x x a b a b a a b a b

state after: 0 0 1 2 3 4 5 ?

diff seq: 0 0 1 2 2 2 2 3 4 5 6

- if the machine could guess right every time, it would match properly
  - avoids "backing up", decides about each character the first time it's seen
- a NDFA matches an input if there is any possible path from start state to a final state.
- it rejects/does not match if there is no path from the start state to a final state.
- how do we make a machine that's always lucky?
  - make a deterministic finite automaton that simulates the NDFA

Egrep: regexpr $\rightarrow$ NDFA $\rightarrow$ DFA

- Example: (a)a(aaa)b

NDFA:

- Convert to DFA by inventing states that represent sets of states of the NDFA:

  - Recognition time is $O(n)$
  - Construction time could be $O(2^n)$
    - because there are $2^n$ subsets of the states
    - newer versions construct states as needed: lazy evaluation
Important ideas from regexps & grep

• tools: let the machine do the work
  - good packaging matters
• notation: makes it easy to say what to do
  - may organize or define implementation
• hacking can make a program faster, sometimes, usually at the price of more complexity

• a better algorithm can make a program go a lot faster

• don’t worry about performance if it doesn’t matter (and it often doesn’t)

• when it does,
  - use the right algorithm
  - use the compiler’s optimization
  - code tune, as a last resort