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:
  - Chris DeCoro, cdecoro@cs, CS 103B, 258-0944
  - Aquinas Habor, aahabor@cs, CS 214, 258-1793

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

• Check out the course web page (CS, not Blackboard)
  - notes, readings and assignments will be posted there
  - Assignment 1 is posted
  - project information is posted

• Do the survey if you haven’t already

Themes

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

• tools
  - where did they come from and why
  - how they have evolved, mutated, decayed
  - how to use them
  - how they work
  - how to build your own

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

• history and culture of programming
(Very) Tentative Outline

Feb 1    regular expressions; grep
Feb 8    scripting languages: Awk & Perl
Feb 15   more scripting: Perl, PHP(?), CGI
Feb 22   Java; object-oriented programming
Mar 1    networking; MySQL; project
Mar 8    user interfaces, Swing

Mar 15   (spring break)

Mar 22   C++
Mar 29   C++, Standard Template Library
Apr 5    Visual Basic; COM, components
Apr 12   XML, web services; .net, C#
Apr 19   Tcl/Tk?, language tools?
Apr 26   ?

May 4-5  project presentations

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\2    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`  \textit{thing} anywhere in string
- `^thing` \textit{thing} at beginning of string
- `thing$` \textit{thing} at end of string
- `^thing$` string that contains only \textit{thing}
- `^$` empty string
- `.` non-empty, i.e., at least 1 char
- `^` matches any string, even empty
- `thing.$` \textit{thing} plus any char at end of string
- `thing\.$` \textit{thing} at end of string
- `\\thing\` \textit{thing}\ anywhere in string
- `[tT]thing` \textit{thing} or \textit{Thing} anywhere in string
- `thing[0-9]` \textit{thing} followed by one digit
- `thing[^0-9]` \textit{thing} followed by a non-digit
- `thing[0-9][^0-9]` \textit{thing} followed by digit, then non-digit
- `thing1.*thing2` \textit{thing1} then any text \textit{then thing2}
- `^thing1.*thing2$` \textit{thing1} at beginning and \textit{thing2} at end

\textbf{egrep: fancier regular expressions}

- `r+` one or more occurrences of \textit{r}
- `r?` zero or one occurrences of \textit{r}
- `r1|r2` \textit{r1} \textit{or r2}
- `(r)` \textit{r} (grouping)

\(([0-9]+\.?[0-9]*\|[\.\[0-9]+/)\(\([\text{Ee}][-+]?[0-9]+\)\)?
Grammar for egrep regular exprs

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

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

\( ([0-9]+\.?[0-9]*|\.?[0-9]+) ([Ee][-+]？[0-9]+)? \)

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, C# ... 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
", 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 regexpr 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

```c
/* 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
```

```
      Vtime  Seconds  Cumsecs  #Calls  msec/call  Name
      52.1  0.37     0.37     4360969  0.0001    matchhere
      16.9  0.12     0.49     4528173  0.0000    _mcount
       11.3  0.08     0.57     31102  0.0026    match
        8.5  0.06     0.63      546  0.11    _read
       16.9  0.05     0.68     31642  0.0016    __memcopy
        1.4  0.01     0.69      546  0.02    __filbuf
        1.4  0.01     0.70     31103  0.0003    fgets
        1.4  0.01     0.71     32429  0.0003    strlen
        0.0  0.00     0.71       1  0.00    main
        0.0  0.00     0.71       1  0.00    grep
        0.0  0.00     0.71       1  0.00    setprogs
        0.0  0.00     0.71     1326  0.000  printf
        ...
```

```
$ wc ../bib
  31102  851820  4460056 ../bib
```

```
$ a.out x ../bib | wc
  1326   39057  207477
```

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

```bash
$ gcc -fprofile-arcs -ftest-coverage grep.c; a.out x ../bib
$ gcov grep.c; cat grep.c.gcov

/* matchhere: search for regexp at beginning of
int matchhere(char *regexp, char *text)
4360969 {
4360969     if (regexp[0] == '\0')
1326         return 1;
4359643     if (regexp[1] == '*')
#####     return matchstar(regexp[0], regexp+2, text);
4359643     if (regexp[0] == '$' && regexp[1] == '\0')
#####     return *text == '\0';
4359643     if (*text!='\0' && (regexp[0]=='.' || regexp[0]=='\0'))
1326         return matchhere(regexp+1, text+1);
4358317     return 0;
}

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
31102 {
31102     if (regexp[0] == '^')
#####     return matchhere(regexp+1, text);
4359643     do { /* must look even if string is empty */
4359643         if (matchhere(regexp, text))
1326             return 1;
4358317     } while (*text++ != '\0');
29776     return 0;
}

• note conservation laws
```

How to make grep faster

• use optimization (cc -O)

• change compilers (icc, 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] == '"0' && regexp[1] == '.*'
        && regexp[2] == 'x")
        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

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

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

- **a^n b^n, 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|aa|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 \(O(2^m)\)
  - because there are \(2^m\) subsets of the states
  - newer versions construct states as needed:
    - lazy evaluation
Important ideas from regexprs & 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