

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

Feb 3	regular expressions; grep
Feb 10	scripting languages: Awk & Perl
Feb 17	more scripting: Perl, PHP(?), CGI
Feb 24	Java; object-oriented programming
Mar 2	Java; networking; databases; project
Mar 9	user interfaces, Swing
Mar 15	(spring break)
Mar 23	C++
Mar 30	C++, Standard Template Library
Apr 6	Visual Basic; COM, components
Apr 13	XML, web services; .net, C# ?
Apr 20	Tcl/Tk?, language tools?
Apr 27	?
May 5-6	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* . [] ^ \$ * + \
- r₁r₂** 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

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

egrep: fancier regular expressions

<code>r+</code>	one or more occurrences of <i>r</i>
<code>r?</code>	zero or one occurrences of <i>r</i>
<code>r₁ r₂</code>	<i>r₁</i> or <i>r₂</i>
<code>(r)</code>	<i>r</i> (grouping)

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

Grammar for egrep regular exprs

r: `c` `.` `^` `$` `[ccc]` `[^ccc]`
`r*` `r+` `r?`
`r1r2`
`r1|r2`
`(r)`

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`, `vim`, ... 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(regex, line)
    print line
```

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



Grep (TPOP, p 226)

```
/* grep: search for regex in file */
int grep(char *regex, 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(regex, 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**

```
/* 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

```
/* 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

```
$ gcc -p grep.c
$ a.out x ../bib >foo
$ prof
%Time 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
7.0 0.05 0.68 31642 0.0016 _memcpy
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. main
0.0 0.00 0.71 1 0. grep
0.0 0.00 0.71 1 0. setprogrnam
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

```
$ lcc -b grep.c; a.out x ../bib >/dev/null; bprint
```

```
int match(char *regexp, char *text)
<31102>{
    if (<31102>regexp[0] == '^')
        return <0>matchhere(regexp+1, text);
    do <4359643>{ /* must look even if string is empty */
        if (<4359643>matchhere(regexp, text))
            return <1326>1;
    <4358317>} while (<4358317>*text++ != '\0');
    return <29776>0;
}
int matchhere(char *regexp, char *text)
<4360969>{
    if (<4360969>regexp[0] == '\0')
        return <1326>1;
    if (<4359643>regexp[1] == '*')
        return <0>matchstar(regexp[0], regexp+2, text);
    if (<4359643>regexp[0] == '$' && <0>regexp[1] == '\0')
        return <0>*text == '\0';
    if (<4359643>*text!='\0' && <4329867>(regexp[0]=='.' ||
    <4329867>regexp[0]==*text))
        return <1326>matchhere(regexp+1, text+1);
    return <4358317>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] != '\\0' && 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

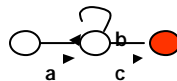
```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
<31102>{
    char *p;
    if (<31102>regexp[0] == '^')
        return <0>matchhere(regexp+1, text);
    if (<31102>regexp[0] != '\\0' && <31102>regexp[0] !=
    '.' && <31102>regexp[1] != '*')
        if (<31102>(p=strchr(text, regexp[0])) == NULL)
            return <29776>0;
    do <1326>{ /* must look even if string is empty */
        if (<1326>matchhere(regexp, p))
            return <1326>1;
    <0>} while (<0>*p++ != '\\0');
    return <0>0;
}
/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
<2652>{
    if (<2652>regexp[0] == '\\0')
        return <1326>1;
    if (<1326>regexp[1] == '*')
        return <0>matchstar(regexp[0], regexp+2, text);
    if (<1326>regexp[0] == '$' && <0>regexp[1] == '\\0')
        return <0>*text == '\\0';
    if (<1326>*text!='\\0' && <1326>(regexp[0]!='.' ||
<1326>regexp[0]==*text))
        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...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 -> 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**



- **$a^n b^n$, 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 (NFA)

RE: `.*ab.*abab`

FSM: 0 1 2 3 4 5 6

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

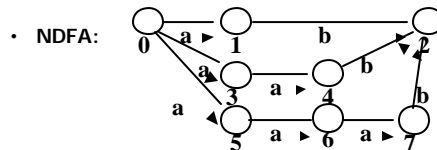
state after: 0 0 1 2 3 4 5 ?

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

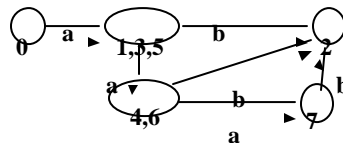
- if the machine could guess right every time, it would match properly
 - avoids "backing up", decides about each character the first time it's seen
- a NFA matches an input if there is any possible path from start state to a final state.
- it rejects/does not match if there is no path from the start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

Egrep: regexpr -> NFA -> DFA

- Example: `(a|aa|aaa)b`



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



- 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 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