Exercise 1-1

TRUE == 0 and FALSE == 1 is contrary to the conventions of C. not\_eof

would be better named is\_eof, with its value set to TRUE in this

example.

Exercise 1-2

int lessequal(char \*s, char \*t)

{

return strcmp(s, t) <= 0;

}

Exercise 1-3

See (listen to) ex1-3.wav.

Exercise 1-4

if (c != 'y' && c != 'Y')

return;

if (length > BUFSIZE)

length = BUFSIZE;

flag = !flag;

quote = (\*line == '"');

bit = val & 1;

Exercise 1-5

There is no guarantee which of val and ch will be read first.

Exercise 1-6

1 1

1 2

2 1

See ex1-6.c. GCC 3.4.6 gives "2 1".

Exercise 1-7

if (!istty(stdin) && !istty(stdout) && !istty(stderr))

return 0;

return retval;

for (k = 0; k < 5; k++) {

scanf("%lf", &dx);

x += dx;

}

Exercise 1-8

The loop increment happens at the top of the loop, so the loop goes from

1 to total, not 0 to total - 1.

The == operator checks equality of references, not string equality. The

getName and userName methods presumably return strings.

for (int count = 0; count < total; count++) {

if (this.getName(count).equals(nametable.userName()))

return true;

}

Exercise 1-9

The entire macro body is not parenthesized. The macro argument c is not

parenthesized. The macro argument c may be evaluated more than once. The

?: operator is superfluous.

Exercise 1-10

#define FT2METER 0.3048

#define METER2FT (1.0 / FT2METER)

#define MI2FT 5280.0

#define MI2KM (MI2FT \* FT2METER / 1000.0)

#define SQMI2SQKM (MI2KM \* MI2KM)

Exercise 1-11

The comment or the code must be incorrect because the method returns

void and the function says it returns a number.

The comment is incomplete; the code actually tests for a even number or

a number that is greater than MAX.

The comment heading the function is fine. The other comments should

refer to "line number" rather than "line counter" for consistency with

the code and the header comment. The two identical comments "increment

line counter" document two different things: increasing line\_number by 1

and increasing it by 2. The comments should be different to note the

difference. However, the function is better written

void write\_message()

{

fprintf(fout, "%d %s\n%d %s\n%d %s\n",

line\_number + 1, HEADER,

line\_number + 2, BODY,

line\_number + 3, TRAILER);

line\_number += 3;

}

and then no comments are necessary.

Exercise 2-1

See qsort-iter.c. The iterative version is not expressed as simply. I

found it desirable to build auxiliary data structures and functions to

handle the stack.

Exercise 2-2

See QuicksortTest.java. Here is some sample output:

Size 1

Objects (Integer): 0.0

Primitives (int): 0.0

Ratio objects / primitives: ?

Size 10

Objects (Integer): 0.0

Primitives (int): 0.0

Ratio objects / primitives: ?

Size 100

Objects (Integer): 0.003

Primitives (int): 0.001

Ratio objects / primitives: 3

Size 1000

Objects (Integer): 0.035

Primitives (int): 0.011

Ratio objects / primitives: 3.182

Size 10000

Objects (Integer): 0.504

Primitives (int): 0.155

Ratio objects / primitives: 3.252

Size 100000

Objects (Integer): 7.135

Primitives (int): 1.822

Ratio objects / primitives: 3.916

Size 1000000

Objects (Integer): 113.429

Primitives (int): 24.997

Ratio objects / primitives: 4.538

With GCJ 3.2.1 the type-conversion penalty is significant, a factor of

three or four.

Exercise 2-3

See qsort-race.c. Here is some sample output:

zero : 8.830 s

ascending : 5.590 s

descending : 5.770 s

onoff : 8.600 s

ramp : 8.890 s

interleaved : 8.690 s

interleaved-reversed: 8.270 s

Exercise 2-4

See slow-sort.c. The time complexity is greater than Omega((2^n) \* n!).

Exercise 2-5

It is not worthwhile to reclaim the space used by a single element,

because up to half of the allocated size of the array is expected to be

unused anyway. A reasonable course would be to decrease the allocated

size by half when the array size drops to a quarter of its capacity.

Exercise 2-6

See growarray.c. The rest of the program now needs to account for the

presence of null elements. For example, see the function printnvtab.

Exercise 2-7

See nvlist.c. insert\_after is much easier than insert\_before. If you

assume that the item after which to insert is in the list, insert\_after

is constant time. I did not get much reuse of the other functions, only

a little of addfront in insert\_after.

Exercise 2-8

See nvlist.c.

Exercise 2-9

See list-generic-c.c, list-generic-c.h, list-generic-c++.h, and

List.java. I used a different style in the different list

implementations. The C version gives the user access to the node

structures that make up the list, while these are hidden in the C++ and

Java versions. This means that the C version supports iteration through

node pointers. The C++ and Java versions only support an index-based get

method, which is inefficient for linked lists. I thought it was going

too far to invent a new iterator interface. C and Java place a large

burden on the programmer by requiring casts to and from void pointers

and Object references. The template approach in C++ is better. Java's

garbage collection simplifies the interface, because there is no need to

provide a way to reclaim memory used by data.

Exercise 2-10

See nvlist.c, list-test-c.c, list-test-c++.cpp, and ListTestJava.java.

Exercise 2-11

See tree.c. Here are the times for searching for every element in a

balanced tree of 50,000 elements:

build tree: 2.2100 s

lookup: 0.0300 s

nrlookup: 0.0300 s

Here for 400,000 elements:

build tree: 19.8800 s

lookup: 0.9800 s

nrlookup: 0.2700 s

It appears recursion gets more expensive the deeper the tree.

Exercise 2-12

See treesort.c. The time complexity is O(n^2) in the worst case,

O(n log(n)) in the average case. It would behave poorly (exhibit

worst-case behavior) when many elements are equal or the list to be

sorted is already sorted in ascending or descending order, which

conditions lead to an unbalanced tree. Here is a sample of sorting

100,000 elements:

treesort: 2.220

our quicksort: 0.900

library quicksort: 0.890

Exercise 2-13

See test\_insert in tree.c. The benchmark tests also serve a testing

purpose.

Exercise 2-14

For NHASH = 127, all these strings hash to 100:

"d", "d0", "d00", "d000", "d0000", "d00000", "d00000r"

and all these hash to 103:

"g", "gU", "gUU", "gUUU", "gUUUU", "gUUUUU", "gUUUUU(", "gUUUUU(x", "gUUUUU(x)"

These strings are built up incrementally, at each step figuring what the

next character must be to keep the hash value the same. The sudden

change from the repeated characters is what happens when the hash

counter overflows an unsigned int. See trivial\_collisions in hash.c.

If NHASH is 31, the hash function degenerates into the hash of the final

character: hash("a") = hash("cba") = hash("hasha"); at least until the

hash counter overflows an unsigned int. Similarly, if NHASH is 62,

strings with the same final character will fall into one of only two

buckets, and so on with integer multiples of MULTIPLIER. This means

short strings with the same final character are likely to collide. See

nhash\_collisions in hash.c.

Exercise 2-15

See apply in hash.c.

Exercise 2-16

See hash-grow.c. x and y are MAX\_AVG\_CHAIN\_LENGTH and

SYMTAB\_GROWTH\_FACTOR.

Exercise 2-17

See hash-point.c. hash\_point is the function that hashes a 2D cartesian

integer point, and the other hash functions are variations on it. I

treat the vector of coordinates in a point as a little string and use

the same hash function from earlier. It generalizes easily to higher

dimensions. The output appears a little less uniform when hashing polar

coordinates of regularly spaced cartesian points. (A consideration with

polar coordinates is whether different representations of the same

point, like (1, 0) and (1, 2 \* pi), should hash to the same value. I

have not attempted to make such points hash to the same value.) This

hash scheme works fine for floating-point coordinates until the points

get too close together and overwhelm the ability of the multiplier to

dissipate them.

Exercise 3-1

See randselect.c. The function genrand simulates the selection algorithm

using a given random number generation function. Using stdlib rand gives

a fairly uniform distribution. Using stdlib rand and throwing out odd

values gives a complex distribution where odd values are more likely

than even. Using a linear congruential generator with made-up constants

is really bad, never producing the numbers 0, 3, 7, 11, .... A generator

that always returns 0 always selects the last element in the list.

Exercise 3-2

See word-savings.c for a measure of how much space is saved storing

duplicate words together. Here is its output for some sample texts.

(Texts use the Project Gutenberg file names. 2ws2410.txt is Julius

Caesar, 76.txt is the Adventures of Huckleberry Finn, kjv10.txt is the

King James Bible, ulyss12.txt is Ulysses. Project Gutenberg boilerplate

has not been removed.)

2ws2410.txt: words: 22756 to 5853 bytes: 126781 to 44182 = -65.15%

76.txt: words: 113342 to 13831 bytes: 579413 to 109226 = -81.15%

kjv10.txt: words: 823156 to 34027 bytes: 4317928 to 272725 = -93.68%

ulyss12.txt: words: 267235 to 52914 bytes: 1517948 to 441433 = -70.92%

The space savings appear to be substantial.

For the tests see markov.c, markov-hash.c, and markov-test.sh. markov.c

was modified to use a constant random seed and report memory use. Here

is a summary of the output of markov-test.sh:

2ws2410.txt:

markov 0m1.022s 615867 bytes.

markov-hash 0m0.973s 612854 bytes.

76.txt:

markov 0m5.535s 2463491 bytes.

markov-hash 0m4.828s 2136714 bytes.

ulyss12.txt:

markov 26m25.064s 6583586 bytes.

markov-hash 0m16.516s 5963145 bytes.

Processing kjv10.txt caused my system to thrash and it hadn't completed

after several hours. The ulyss12.txt results are anomalous. Presumably

its memory use was just enough to cause markov to thrash.

markov-hash is a little faster and uses a little less memory than

markov, but not nearly as much as I would have expected from measuring

the degree of word repetition in the texts. My guess as to why

markov-hash is not much faster is that the maximum output length, 10,000

words, is small in comparison to the size of the inputs. The faster

hashing doesn't get much of a chance to contribute. As for the memory

use, that can be explained by the fact that the storage of strings is a

small part of the total memory used by the program. Arrays and

structures take up the majority. Here is the full memory use output from

76.txt:

markov-76.txt:

Size of state hash table: 16372 bytes.

Size of State structures: 960960 bytes.

Size of Suffix structures: 906744 bytes.

Size of all strings: 579415 bytes.

Total 2463491 bytes.

markov-hash-76.txt:

Size of state hash table: 16372 bytes.

Size of string hash table: 32764 bytes.

Size of State structures: 960960 bytes.

Size of Suffix structures: 906744 bytes.

Size of String structures: 110648 bytes.

Size of all strings: 109226 bytes.

Total 2136714 bytes.

While the size of string storage was reduced 56%, string storage

accounted for only 24% of the total storage used by markov, leading to

only a 13% decrease in total memory usage.

Exercise 3-3

See markov-nosentinel.c. This version is clumsier, requiring storage of

the initial and final prefixes. A separate step to print the initial

prefix is necessary before starting the Markov algorithm. The most

obtrusive change is in the termination condition. I originally thought

to use a lookup result of NULL (no suffix for the given prefix), but

that gave anomalous results for this interesting test case:

echo 1 1 1 | ./markov

echo 1 1 1 | ./markov-nosentinel

markov stops after generating a few words, while markov-nosentinel goes

on for MAXGEN. The reason is that in the generated hash table there is

no prefix without a suffix. I revised the program so that a lookup

result of NULL is still an unequivocal exit condition, but additionally

if a prefix is equal to the final prefix then the program gets a random

chance to stop, with a probability the same as a sentinel nonword would

have. This case is the only reason for storing the final prefix.

Exercise 3-4

See Markov.java.

Exercise 3-5

See markov++.c. The makefile automatically generates several versions

from one source file by overriding types with preprocessor definitions.

My machine didn't have a hash\_map structure. I tried using a deque,

list, vector, and array for the prefixes and a list and vector for

suffix lists, leading to eight different versions in all. All the

different versions were run by the test structure in markov++-test.sh.

Here are the results. Program names are in the form

markov++-$(queue\_type)-$(list\_type), where $(queue\_type) is the type of

a prefix and $(list\_type) is the type of a suffix list.

markov++-array-list 0m6.966s

markov++-array-vector 0m6.452s

markov++-deque-list 2m13.120s

markov++-deque-vector 1m23.733s (same as authors' version)

markov++-list-list 0m11.438s

markov++-list-vector 0m9.417s

markov++-vector-list 0m9.828s

markov++-vector-vector 0m8.690s

Using an array for the prefix list appears to be the fastest, though

this advantage would likely be lost if the prefix length were increased.

The deque had awful performance, possible because it induced swapping.

Exercise 3-6

See markov++-simple.cpp. In style it is a mixture of the C version and

the C++ with STL version. More intelligence is built into the classes

representing prefixes and suffixes because of the lack of intelligence

of their containers. This version is the fastest.

markov++-simple 0m4.795s

Exercise 3-7

See markov-npref.awk and markov-npref.pl.

$ time awk -f markov.awk < 2ws2410.txt > /dev/null

real 0m6.994s

$ time awk -f markov-npref.awk < 2ws2410.txt > /dev/null

real 0m20.336s

$ time perl markov.pl < 2ws2410.txt > /dev/null

real 0m6.356s

$ time perl markov-npref.pl < 2ws2410.txt > /dev/null

real 0m7.762s

The penalty for generality is quite tolerable in Perl.

Exercise 3-8

See markov.py. In flavor it is similar to the Awk and Perl versions, in

fact I referred to the Perl version while writing it.

$ time python markov.py < 2ws2410.txt > /dev/null

real 0m12.928s

It's about twice as slow as the two-word-prefix Awk and Perl versions,

though faster than the general Awk version and still slower than the

general Perl version.

Exercise 4-1

Split all at once when a field is read. This is the easiest approach. It

has the benefit of simplicity. The downside is that unnecessary work may

be done, when fields are split that aren't used, or when no fields from

a line need to be used. This is implemented in csv-split-0.c.

Split all at once when any field is requested. This is only a little

more difficult and complex than the previous approach. Its only

potential performance improvement will come when lines are read but not

fields from them are used. I think this is an unlikely usage. This is

implemented in csv-split-1.c.

Split only the field requested. This likely has the best performance

when only a few fields from each line are required. If all fields are

required, or the same field is requested more than once, it is likely to

be slower than the first two options. It would be harder than the others

to implement. This is implemented in csv-split-2.c.

Split up to the field requested. This would not incur a penalty for

requesting the same field over and over. It would probably be a little

easier to implement than the third option. This is implemented in

csv-split-3.c.

See csv-split-test.c for the test driver code and genquote.pl for the

code that generates the test data. Each splitting technique is used in

four different scenarios: retrieving all fields, retrieving no fields,

retrieving only the first field, and retrieving a single field more than

once. Here are the results.

csv-split-test-0

all: 1.11 s

none: 0.97 s

first: 0.99 s

repeated: 0.97 s

csv-split-test-1

all: 0.99 s

none: 0.79 s

first: 0.97 s

repeated: 0.97 s

csv-split-test-2

all: 3.20 s

none: 0.79 s

first: 0.94 s

repeated: 1.25 s

csv-split-test-3

all: 1.06 s

none: 0.79 s

first: 0.83 s

repeated: 1.01 s

The first and last options (splitting on read or splitting up to the

required field) are the best for a combination of performance and

simplicity.

Exercise 4-2

See csvgetline2-a.c, csvgetline2-b.c, and csvgetline2-c.c.

For allowing an arbitrary class of characters, in csvgetline2-a.c, there

is no run-time interface. The character class is hard-coded by the

programmer at compile time. Observe that " is accepted as a separator,

though its use is strange because it's also the quote character. Usually

it's interpreted as quote except in weird cases like abc"def, which is

split to "abc", "def"; or "abc"def"ghi, which is split to "abcdef",

"ghi".

For allowing different separators for different fields, I created a new

function csvfieldseps that takes a string. Character 0 of the string is

the separator that follows field 0, and so on. The last character is

reused if necessary. You can call it with an empty string to get the

default separator, a comma.

For allowing separation by a regular expression, I created a new

function csvfieldsep that sets the expression. The default expression

splits on commas but also consumes padding spaces.

Exercise 4-3

See csvgetline2-init.c. The role of reset is the same as it was in the

original implementation. It deallocates the internal buffers and sets

them up to be reallocated.

Exercise 4-4

See csvw.c and csvw.h. The library writes only to a FILE. It is a

low-level library, writing entries one at a time. As much as possible,

it operates directly on the output stream so as not to have to allocate

memory. It quotes strings only if they need it, using a function that

examines a string to see if it is necessary.

Exercise 4-5

See csvgetlinec++.c.

Exercise 4-6

See Csv.java. For the benchmarks I used the same genquote.pl test data.

$ ./genquote.pl > quotes-big.csv

$ time ./csvgetline2 < quotes-big.csv > /dev/null

real 0m2.720s

user 0m2.670s

sys 0m0.040s

$ time ./csvgetlinec++ < quotes-big.csv > /dev/null

real 0m15.273s

user 0m15.200s

sys 0m0.060s

$ time ./Csv < quotes-big.csv > /dev/null

real 0m23.342s

user 0m15.690s

sys 0m1.000s

The C version is the fastest, followed by the C++ version and the Java

version. The C version is the least clear because of all the memory

allocation in the midst of the program logic, and the Java version is

the clearest. The C version is the most robust, as it will be able to

recover from a memory allocation error. The C++ version appears to have

a bug, which is that if a line ends with a quoted field, the line

if (s[j] == '"' && s[++j] != '"') {

will read a character past the end of the string. (This error is

acknowledged on the book's web site.) This logic threw an exception when

I initially copied it to the Java version. Both the C and C++ versions

allow setting a string for the field separator, but they assume

elsewhere in the code that the length of the separator is 1

(i = j + 1, p = sepp + 1). I fixed this in the Java version.

Exercise 4-7

See csvgetline++-iter.c. I didn't know in what way the class was meant

to become an iterator: as an iterator over lines, over fields within a

line, or both. I made it an iterator over input lines by making Csv a

simple extension of istream\_iterator.

Exercise 4-8

See csvgetline2-new.c.

Exercise 5-1

See safe-malloc.c and malloc-mistakes.c for the driver code.

malloc-mistakes takes an integer command-line argument specifying a

memory test to do. Here is the output:

$ ./malloc-mistakes

Usage: ./malloc-mistakes testno

testno is 0-3 for a malloc test to run.

0: overwrite beginning of block.

1: overwrite end of block.

2: free uninitialized pointer.

3: double free.

$ ./malloc-mistakes 0

in-use malloc header scrambled (head 0x804e250, tail 0x804a270, user pointer 0x804e260)

Aborted

$ ./malloc-mistakes 1

in-use malloc tailer scrambled (head 0x804e250, tail 0x804e270, user pointer 0x804e260)

Aborted

$ ./malloc-mistakes 2

in-use malloc header scrambled (head 0x8048ea4, tail 0x8049088, user pointer 0x8048eb4)

Aborted

$ ./malloc-mistakes 3

in-use malloc tailer scrambled (head 0x804e250, tail 0x804e270, user pointer 0x804e260)

Aborted

Exercise 5-2

See strings.c.

Exercise 5-3

See vis.c.

Exercise 5-4

If the input is \X0A the output is the same: \X0A. That's because all

the characters in the input are printing characters. The output would be

unambiguous if one of the characters in the escape sequence was itself

escaped. If we also escape \ then the output would be \X5C0A.

Exercise 5-5

See vis.c. Use -w to set the column width. Use -x to omit non-printing

characters. vis could also display non-printing characters as . or some

other character so they still take up one character visually.

Exercise 6-1

See ex6-1.cpp for some test code.

(a) The original function fails because it decrements n before

multiplying it with fac. This means that when n is 1 at the top of the

loop, it is 0 inside the loop, and the function calculates 0 for any

argument other than 0. It will also take a long time to run and return

nonsense if its argument is negative.

int factorial(int n)

{

int fac;

fac = 1;

for ( ; n > 0; n--)

fac \*= n;

return fac;

}

(b) The original code fails with a zero-length string because it tests

for the end of the string only after running one iteration. This means

that it will print all characters following the zero-length string up to

the first '\0' byte.

for (i = 0; s[i] != '\0'; i++) {

putchar(s[i]);

putchar('\n');

}

(c) The original function fails to copy the null terminator to the

destination string.

void strcpy(char \*dest, char \*src)

{

int i;

for (i = 0; src[i] != '\0'; i++)

dest[i] = src[i];

dest[i] = '\0';

}

(d) The original function fails to null-terminate the destination,

whether the loop finishes because n drops to zero or because the source

runs out.

void strncpy(char \*t, char \*s, int n)

{

int i;

for (i = 0; n > 0 && s[i] != '\0'; i++, n--)

t[i] = s[i];

t[i] = '\0';

}

(e) The original code wrongly prints says that i is smaller than j when

i and j are equal.

if (i > j)

printf("%d is greater than %d.\n", i, j);

else if (i < j)

printf("%d is smaller than %d.\n", i, j);

(f) The original code puts the dividing line in the wrong place; 'M' is

in the first half of the alphabet.

if (c >= 'A' && c <= 'Z') {

if (c <= 'M')

cout << "first half of alphabet";

else

cout << "second half of alphabet";

}

Exercise 6-2

(a)

January 1, 2000

December 31, 1999

January 2, 2000

February 29, 2000 (should be a leap year)

(b) See test-ctime.c. The test code round-trips times near the Y2K

boundary, checking that time\_t values are unchanged after passing

through the chain ctime -> strptime -> mktime. It also checks that all

strings returned by ctime have a length of 25, to catch the likely error

or rendering the year 2000 as "19100".

I wouldn't try to have code attempt to detect a flawed implementation at

runtime. It would be possible to encapsulate ctime in a function that

did tests like I have described at every call, but if you're going to do

that you may as well include a correct implementation of ctime with your

program. If any implementation bugs are suspected, that's what you have

to do anyway, if you want to do more than halt the program on failure.

(c) Because the output format is rigidly specified, I would precalculate

the output for January 2000 by hand or with another program. Then the

output of the program and the precomputed calendar could simply be

compared bytewise.

(d) The 32-bit time\_t overflow in 2038. This can be tested with code

like

time\_t t = 0x7FFFFFFF;

time\_t u = t + 1;

if (u <= t) {

printf("Time went backwards.\n");

printf("%lu: %s", t, ctime(&t));

printf("%lu: %s", u, ctime(&u));

exit(1);

}

On my machine it produces

Time went backwards.

2147483647: Mon Jan 18 20:14:07 2038

2147483648: Fri Dec 13 13:45:52 1901

Exercise 6-3

I would run it on an empty file and see that it produced no output. I

would run it on 256 files, each containing a single distinct byte. I

would run take a sample file and check that freq produces the same

results for the file and its reverse (may catch something like \r\n

being translated to \n). In each of these cases I would run freq's

output through another program to check for nonprinting characters;

there should be none.

Exercise 6-4

See freq.c. Use the -t option to select the data type used: char, int,

or float. The code isn't super elegant: you have to define four

functions and edit a few data structures to add a new handled type. I

tried solving the problem using C++ and a templated class, but ran into

trouble trying to use polymorphism of the types in an STL container.

Exercise 6-5

See printf-test.c. The main mechanical aid I used was having the program

talk to itself through a pipe. vfprintf writes to the pipe, then the

program reads from it and checks what it reads against the expected

output. I made up the test cases by hand.

Exercise 6-6

See mem-test.c.

Exercise 6-7

See mem-test.c.

Exercise 6-8

When testing the results of the numerical routines for equality, in most

cases it will be necessary to accept any values that are within some

threshold of the expected value. These tests check mathematical

identities and boundary values. In general, there should be precomputed

results of the functions at a few arbitrarily chosen values; this will

be especially helpful for finding regressions. In a high-precision

numerical library, it may be a requirement that certain results should

be as accurate as the numeric type allows; in this case it will be

appropriate to test for strict equality.

sin: Test that sin(0) == sin(pi) == sin(-pi) == 0.0. Test that

sin(pi/2) == sin(-pi/2) == pi. For a few test values, test that

sin(x) == sin(x + 2\*pi). Compare a few test values against precomputed

results.

cos: As with sin, suitably adjusted. Check that sin(x) == cos(x - pi/2)

for a few test values.

tan: Check that tan(pi/2) and tan(-pi/2) raise domain errors. Test that

tan(x) == sin(x) / cos(x).

asin: For a few test values, check that asin(sin(x)) == x. Check also

that sin(asin(x)) == x, using only test values in (-pi/2, pi/2). Test

against some precomputed results. Check that values outside

(-pi/2, pi/2) raise a domain error.

acos: As with asin.

atan: As with asin and acos.

atan2: For x and (nonzero) y of equal signs, check that atan2(x, y) ==

atan(x / y). Check that atan2(1.0, 0.0) == 0.0, atan2(0.0, 1.0) == pi/2,

atan2(-1.0, 0.0) == pi, and atan2(0.0, -1.0) == 3\*pi/2. Check also that

atan2(x, 0.0) == 0.0 for a few positive test values.

sinh: Test that sinh(x) == -sinh(-x). Test some precomputed values.

cosh: Test that cosh(x) == cosh(-x). Test some precomputed values.

tanh: Test that tanh(x) == sinh(x) / cosh(x).

exp: Test against some precomputed results.

log: Test that log(-1.0) raises a domain error. Test that

log(exp(x)) == x for any x and that exp(log(x)) == x for x in

(0.0, infinity).

pow: Test that pow(x, 0.0) == 1.0 for nonzero x and pow(0.0, x) == 0.0

for positive x. Test that pow(-x, 2) == pow(x, 2) and

pow(-x, 3) == -pow(x, 3) for integral x. Test that

pow(pow(x, y), 1/y) == pow(pow(x, 1/y), y) == x for nonintegral x and

positive y. Test that pow(x) raises a domain error when x is negative

and not an integer.

sqrt: Test that sqrt(x) == pow(x, 0.5). Test that sqrt(x) \* sqrt(x) ==

x. Test that sqrt(x) raises a domain error when x is negative.

ceil: Check that ceil(1.0) == 1.0, ceil(1.2) == 2.0, ceil(1.5) == 2.0,

and ceil(-0.5) == 0.0.

floor: Check that floor(1.0) == 1.0, floor(1.2) == 1.0,

floor(1.5) == 1.0, and floor(-0.5) == -1.0.

fabs: Check that abs(x) == x for positive x and abs(x) == -x for

negative x. Also abs(0.0) == 0.0. In this case equality should be exact.

ldexp: Check that ldexp(x, n) == x \* pow(2, n) for some test values.

frexp: Check that frexp(0.5, &exp) == 0.5 and exp == 0. Check that

frexp(1.0, &exp) == 1.0 and exp == 1. Check that after

frac = frexp(x, &exp), ldexp(frac, exp) == x.

modf: Check that after fp = modf(x, &ip), fp and ip have the same sign

as x, and fp + ip == x with strict equality.

fmod: Check that Check that fmod(x, 1.0) == modf(x, &dummy). For several

test values of x and y of all sign combinations, check after running

r = fmod(x, y);

modf(x / y, &ip);

that x == y \* ip + r;

Exercise 6-9

strcpy: Create a buffer big enough to hold a test string with space on

both sides. Initialize the buffer with some known pattern. Copy a test

string into the middle of the buffer. Check that the result string is

null terminated and that all its characters were copied. Check that all

other bytes in the buffer are untouched. Repeat for string lengths 0, 1,

2, 3, 4, 5, 6, 7, 8, 9, 2^n - 1, 2^n, and 2^n + 1 for n up to 16; for

destination offsets from 0 to 10; and for source offsets (indices into

the source string) from 0 to 10, or up to the string length.

strncpy: As with strcpy, with the addition of testing different values

of n, from 0 up to the string length \* 2. There should always be exactly

n characters written, with null padding if n is greater than the string

length + 1. The written string must always be null terminated.

strcat: Use the strategy used for memcpy. Build a slow, straightforward,

correct strcat. If memcpy is trusted you could use

memcpy(s + strlen(s), t, strlen(t) + 1) as the basis of it (you would

have to return s rather than what memcpy returned). Create two test

buffers. Copy a string into the middle of the buffer, then concatenate a

string to the end of it. Compare the output of strcat with that of the

independent implementation. Use source and destination offsets from 0 to

10, and string lengths 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 2^n - 1, 2^n, and

2^n + 1 for n up to 16.

strncat: As with strcat, with the addition of testing different values

of n, from 0 up to the greater string length \* 2.

strcmp: Check the equality of non-equality of test strings with source

and destination offsets ranging from 0 to 10. Include in the test data

some strings that are equal up to the length of the shorter string.

strncmp: As with strcmp, with the addition of testing different values

of n, from 0 up to the greater string length \* 2. Include in the test

data some strings that are equal up to the length of the shorter string.

strchr: Build test strings containing the search character in different

places and numbers of occurrences. If c == '#', use strings like

"", "#", "a", "#a", "a#", "##", "#a#", "#abcde", "abcde#",

"#abcde#", "#ab#cde", "abc#de#", "a#b#cde", "abc#d#e"

Test also a value of c larger than an char can hold; it should give the

same results as if it was (char) c.

strrchr: As with strchr, using the same test data.

strspn: Use various sizes for the acceptable set: 0, 1, 2, 10, 100, 255.

Use test strings that start with 0, 1, 2, 3, and 4 characters from the

acceptable set, and strings that consist entirely of such characters.

Check that strspn(s, "") == 0.

strcspn: This function is like strchr. Therefore use similar test

patterns as the strchr test, filling in characters from the forbidden

set in place of '#'. Use various sizes for the forbidden set: 0, 1, 2,

10, 100, 255. Check that strcspn(s, "") == strlen(s). Define a function

comp that produces the set complement of a set of characters. Then check

that strspn(s, chars) == strcspn(s, comp(chars)).

strpbrk: Test this function against strcspn using the same test data. If

s[strcspn(s, chars)] == '\0', then strpbrk(s, chars) should be NULL.

Otherwise it should be the case that

strcspn(s, chars) == strpbrk(s, chars) - s.

strstr: Test against strchr using one-character test strings. Build test

strings containing the search string in different places and numbers of

occurrences. If the search string is "str", use strings like

"", "str", "strstr", "#####", "#str", "str#", "#strstr",

"str#str", "strstr#", "#str#str", "str#str#", "#str#str#"

Test that strstr(s, "") == strstr(s, s) == s.

strlen: Test strings of length 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 2^n - 1,

2^n, and 2^n + 1 for n up to 16.

strtok: For a set of test strings, execute the sequence

p = s + strspn(s, chars);

n = strcspn(p, chars);

q = strtok(s, chars);

After this it should be the case that p == q and n == strlen(q), unless

q == NULL, in which case \*p should be '\0' and n should be 0. This

process is then repeated, using p + n + 1 for s in the lines with strspn

and strcspn and NULL for s in the line with strtok. The test data should

be the same as for strcspn, with a few concatenations of the test

strings added. Test different forbidden set sizes as with strcspn.

Include test cases using a concatenation of test strings from rounds of

the strcspn tests using different forbidden sets.

Exercise 6-10

I set out to try to crash Nmap or Zenmap 4.76. Zenmap seemed an easier

target, so I started there. It had recently been made more robust in its

parsing of XML files (catching any errors during parsing and showing a

dialog instead of crashing) so plain syntax errors didn't work. I tried

introducing invalid numeric strings in attributes, hoping they would

make it past the parser and cause a conversion error elsewhere, but I

couldn't find any that weren't checked somehow.

Nmap has limited options for reading an input file. There is the -iL

option for reading a list of hosts. The parser for that is fairly simple

and is careful about checking string bounds. It appropriately rejects a

binary file (the nmap executable). Another possibility was the --resume

option that parses a partial scan log and starts the scan where it left

off. Again the parser is careful about checking bounds. I tried the

trick of changing the log file name in the log itself, both to another

file name and to "-" for standard input, but neither of those worked.

Finally I thought of a DNS message parsing vulnerability taught to me by

my professor Steve Beaty. As a form of compression DNS messages may

contain pointers to refer back to themselves, and these pointers can be

arranged in a loop. A naive parser will follow the loop and hang or

overflow the stack. I had previously found Nmap's parallel reverse-DNS

resolver not to be vulnerable. However I found that the parser in NSE's

dns module is vulnerable.

I created a file, dns.bytes, containing a DNS message whose hexadecimal

representation is

1234 8180 0001 0000 0000 0000 C00C

Broken down, this is

1234 The transaction ID, could be anything.

8180 Flags indicating a response, copied from a packet capture.

0001 Number of questions.

0000 Number of answers.

0000 Number of authority RRs.

0000 Number of additional RRs.

C00C Ostensibly the question; really a message pointing to itself.

The top two bits of C00C are 11, meaning that the rest of the two bytes

are a pointer. Ignoring those two bits gives the pointer: 000C, 12

decimal, the index of the pointer itself.

I set up a pseudo-DNS server on localhost with

ncat -u -l localhost 53 < dns.bytes

I then ran

nmap -d -sP --script=asn-query.nse --script-args dns=localhost scanme.nmap.org

-d is necessary to make an error message show up, -sP does a ping scan

rather than a full port scan, asn-query.nse is a script that uses the

dns module, and --script-args dns=localhost makes the script use the

pseudo-server. This caused the following output:

SCRIPT ENGINE: Will run /usr/share/nmap/scripts/asn-query.nse against 64.13.134.52

SCRIPT ENGINE: Running scripts.

SCRIPT ENGINE: Runlevel: 1.000000

Initiating SCRIPT ENGINE at 15:34

SCRIPT ENGINE: /usr/share/nmap/nselib/dns.lua:486: stack overflow

Completed SCRIPT ENGINE at 15:34, 0.09s elapsed

SCRIPT ENGINE: Script scanning completed.

The effect of this is small: it doesn't crash Nmap, only one script (any

other scripts keep running). Because the overflow is in the embedded Lua

environment there's little chance of it being an exploitable

vulnerability. Still, it's a crash caused only by external input.

Exercise 7-1

See spam-1.c and spam-2.c. Here are average timing values for 10 runs of

the two programs on Ulysses.

spam-1:

real 0m0.470s

user 0m0.362s

sys 0m0.110s

spam-2:

real 0m0.586s

user 0m0.463s

sys 0m0.121s

The two-character prefix version is slower. This may be because of extra

lookup overhead that would be diminished with a larger pattern table.

Exercise 7-2

See timer.c, timer.h, and timer-test.c. This timer is meant to be linked

with an application. It is designed for ease of use, not generality.

Only one timer at a time is supported.

The timer output matches a wall clock as long as the machine is not

doing much. Other activity on the machine doesn't have much of an effect

on the timer time but makes programs slower in real time. On a quiet

machine,

$ time ./timer-test

0.83 s

real 0m0.862s

user 0m0.360s

sys 0m0.480s

Then running yes simultaneously to saturate the CPU,

$ time ./timer-test

0.89 s

real 0m2.440s

user 0m0.430s

sys 0m0.480s

Exercise 7-3

48,350,000 - 46,180,000 is 2,170,000, or 217 \* 10,000. The fixed testing

message must not have contained any of the 217 spam patterns. When

strstr doesn't find a match it called strchr once more than strncmp.

Exercise 7-4

See memset.c. Here is its output.

size 832040

offset 0 1 2 3

library memset: 0.04 0.02 0.01 0.02

byte-at-a-time memset: 0.1 0.1 0.09 0.1

word-at-a-time memset: 0.03 0.03 0.04 0.03

size 1346269

offset 0 1 2 3

library memset: 0.04 0.03 0.03 0.03

byte-at-a-time memset: 0.16 0.15 0.16 0.16

word-at-a-time memset: 0.05 0.05 0.05 0.05

size 2178309

offset 0 1 2 3

library memset: 0.07 0.05 0.04 0.05

byte-at-a-time memset: 0.26 0.25 0.25 0.26

word-at-a-time memset: 0.08 0.08 0.09 0.08

size 3524578

offset 0 1 2 3

library memset: 0.11 0.08 0.07 0.07

byte-at-a-time memset: 0.41 0.42 0.41 0.41

word-at-a-time memset: 0.13 0.13 0.13 0.13

The straightforward byte-at-a-time memset takes around 5.5 times as long

as the library version. The optimized memset takes only 1.5 times as

long.

I patched fast\_memset into mem-test.c to make sure it was correct. This

found a few bugs in my first implementation.

Exercise 7-5

See smalloc.c. The threshold at which to start allocating with malloc is

ideally dependent on the strings to be stored. If it is too large, some

space will be wasted in the fixed-size blocks; if too large, all space

in the blocks will be wasted, and smalloc acts like malloc with more

overhead. The program strlen.awk gets the mean and standard deviation of

lengths of strings as used by the Markov program. Its output on the

Markov test texts is

2ws2410.txt:

N = 23898

Mean 4.56

Standard deviation 0.0230

76.txt:

N = 115786

Mean 4.13

Standard deviation 0.0114

kjv10.txt:

N = 848400

Mean 4.24

Standard deviation 0.0035

ulyss12.txt:

N = 274944

Mean 4.67

Standard deviation 0.0070

I chose a threshold of 12 based on gut feeling. That appears to be a

reasonable choice. It's about double the mean string length when

including the null terminator.

Exercise 7-6

See op-cost.c. Measurements from five different host/OS/compiler

combinations are in op-cost.txt. I built the program in each case with

the default compiler options. I adjusted NUM\_ITERATIONS upwards on

faster machines.

In all cases, division and modulus are the slowest integer operations.

Likewise with floating-point division. The Mac OS X single-precision

floating point divide is faster than that of Windows XP on the same

hardware. Windows XP had very slow function calls, perhaps as a result

of debugging checks. Windows has strange I/O measurements: the operation

that is fastest on other platforms, fputs, is slowest there; the slowest

operation on other platforms, fgets, is there the fastest, though still

slower than the slowest operation on Mac OS X on the same hardware. All

platforms except for Windows appear to optimize strcmp(s, s).

String/number conversions kept pretty much the same ordering on all

platforms. Same with the math functions.

There seems to be some problem with the timer calibration technique I

used, that of timing an empty loop. Some of the operations have a

negative duration after the calibration value is subtracted.

Exercise 7-7

See op-cost-c++.cpp.

Exercise 7-8

See OpCost.java. Java doesn't have a preprocessor, which had been

convenient in the C and C++ timing code. So I just used cpp to transform

OpCost.java.in to OpCost.java before compiling. The makefile does it

automatically.

Exercise 8-1

See conditional.c. I compiled it with GCC 3.4.6, with and without

optimization. Using "objdump -d -S conditional" I saw that the compiler

generates code for

if (debug())

but does not generate code for

if (DEBUG)

if (0)

The compiler appears always to check syntax. Uncomment the "syntax

error" lines to see.

I got the same results with GCC 4.3.1 when compiling with the default

options. However if I compile with moderate optimization (-O2), the

compiler omits code for

if (debug())

Exercise 8-2

See crlftolf.pl and lftocrlf.pl. For testing see crlf.txt and lf.txt.

These two files have the same contents except for line terminators. They

have a few potentially tricky test cases, which are: a newline at the

beginning of the file, a bare carriage return in the middle of a line,

two consecutive newlines, and no newline at the end of the file.

crlftolf.pl should turn crlf.txt to lf.txt and lftocrlf.pl should turn

lf.txt into crlf.txt:

./crlftolf.pl crlf.txt > tmp.txt

cmp lf.txt tmp.txt

./lftocrlf.pl lf.txt > tmp.txt

cmp crlf.txt tmp.txt

Running crlftolf.pl on lf.txt and running lftocrlf.pl on crlf.txt should

be no-ops. In particular, lftocrlf.pl should not turn \r\n to \r\r\n.

./crlftolf.pl lf.txt > tmp.txt

cmp lf.txt tmp.txt

./lftocrlf.pl crlf.txt > tmp.txt

cmp crlf.txt tmp.txt

Exercise 9-1

See pack.c and pack-test.c. I used capital letters to signify a signed

item. pack-test unpacks a file with packed signed and unsigned values,

checks that the unpacked values are correct, and packs them back into

another file. The two files should be the same. I ran the program on a

32-bit and a 64-bit machine; all values were unpacked correctly and all

input and output files were the same.

Exercise 9-2

See pack.c and pack-test.c. I chose $ as the string format character.

Instead of embedding the string's length in the format string, it is

passed as another argument just before the string pointer. On unpacking,

the length is given again, but this time it's a maximum number of bytes

to store in the target buffer.

Exercise 9-3

See pack-c++.cpp. I may have misunderstood the exercise. Of the three

functions mentioned, I did not have to modify pack and unpack, only

receive. readpacket is wrapped by getpacket, which returns a packet

object. The packet class's pure virtual functions pack and unpack are

implemented in subclasses packet\_type1 and packet\_type2. The receive

loop is a bit simpler, but now there is a switch in getpacket like the

one that was avoided in receive by the use of function pointers. One

advantage is that each packet type's pack member function can operate on

the packet's data members, so all the pack functions have the same

signature.

Exercise 9-4

See printf.c and test-printf.sh.

I thought about using parse\_printf\_format to get the argument types, and

then pass converted arguments to printf one at a time. I would parse the

format string to read past the first format specifier, then pass zero,

one, two, or three command-line arguments to printf called with that

fragment of the format string. (Zero for %%, two or three for \* width or

precision.) But that way seems fraught with peril. There is a danger

that my parsing and that of the C library will not match, precisely the

problem I was trying to guard against by using parse\_printf\_format. Also

there is the matter of the p and n conversion specifiers--what sense do

those make on the command line? The question is how to write the program

without reimplementing printf.

What we do is parse each printf format specifier individually

(parse\_printf\_conversion), then rewrite it (render\_printf\_conversion)

and pass the rewritten format to printf. Any conversion that is not

understood is copied to the output with a warning. The user's input

always passes through this filter; it guarantees we never send anything

untrusted directly to printf. It also allows us to omit support for

problematic formats like p and n. Each rewritten format then consumes

zero, one, two, or three arguments.

Exercise 9-5

See decimal-format.c and decimal-format-test.sh. A simple reference

implementation is in DecimalFormat.java.

This exercise was a challenge because of a lack of a clear

specification. I first wrote a version that turned #s into blanks, then

checked the Java documentation to see what other features were

necessary. To my surprise, Java's DecimalFormat doesn't format #s as

blanks. (Indeed there is no facility for variable blank-padding.) That

required a reworking of the program.

I then followed the Java documentation and designed my program to handle

prefixes, suffixes, and grouping like Java does. I started writing

support for scientific notation and wrote tests using examples from the

Java documentation. But Java's DecimalFormat failed one of the tests!

The DecimalFormat JavaDoc says: "The number of significant digits in the

mantissa is the sum of the minimum integer and maximum fraction digits,

and is unaffected by the maximum integer digits. For example, 12345

formatted with '##0.##E0' is '12.3E3'." But the result of formatting

that was "12.345E3" on two different Sun Java implementations I tested.

(Tests also found bugs in the DecimalFormat of GCJ 4.3.1 and 3.2.1.)

I also found that single quotes in a prefix and suffix don't quote just

the character that follows, but everything up to the next single quote.

(Except in the case of "''", which becomes a single single quote.) That,

combined with the silly percent/per-mille features, made me give up on

the overburdened specification.

I decided to implement my own DecimalFormat, as I pictured it before I

got into the Java specification. This one doesn't do prefixes, suffixes,

negative patterns, or any of that, but it does do blank padding, which

is one of the things I find most useful about printf.

Exercise 9-6

See match-test.c. Output lines come in pairs; the first line shows the

string searched for and the string search, the second compares how many

matches per second match did versus how many strstr did.

"" in ""

match / strstr = 1074855788 / 7420000 = 144.8593

"abc" in ""

match / strstr = 800000 / 7380000 = 0.1084

"xyz" in ""

match / strstr = 800000 / 7340000 = 0.1090

"abc%%%%%%%%%%" in ""

match / strstr = 800000 / 7390000 = 0.1083

"" in "abc"

match / strstr = 1060000 / 7410000 = 0.1430

"abc" in "abc"

match / strstr = 316832 / 7380000 = 0.0429

"xyz" in "abc"

match / strstr = 233010 / 7410000 = 0.0314

"abc%%%%%%%%%%" in "abc"

match / strstr = 150943 / 7440000 = 0.0203

"" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 1050000 / 7440000 = 0.1411

"abc" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 316832 / 7380000 = 0.0429

"xyz" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 31008 / 7390000 = 0.0042

"abc%%%%%%%%%%" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 105769 / 7370000 = 0.0144

"" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 1050000 / 7390000 = 0.1421

"abc" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 51724 / 7440000 = 0.0070

"xyz" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 31008 / 7380000 = 0.0042

"abc%%%%%%%%%%" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

match / strstr = 41322 / 7420000 = 0.0056

"" in "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%abc"

match / strstr = 1060000 / 7410000 = 0.1430

"abc" in "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%abc"

match / strstr = 31496 / 7410000 = 0.0043

"xyz" in "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%abc"

match / strstr = 30769 / 7420000 = 0.0041

"abc%%%%%%%%%%" in "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%abc"

match / strstr = 29197 / 7410000 = 0.0039

match was usually between 0.4% and 10% as fast as strstr. In matching

the empty string against the empty string, however, it was 145 times

faster.

Exercise 9-7

See match-test.c. Output is as with exercise 9-6.

"" in ""

matchhere / matchhere\_iter = 1820000 / 1820000 = 1.0000

"" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 1820000 / 1820000 = 1.0000

"" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 1810000 / 1820000 = 0.9945

"abc" in ""

matchhere / matchhere\_iter = 1340000 / 1310000 = 1.0229

"abc" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 356436 / 440000 = 0.8101

"abc" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 1100000 / 1090000 = 1.0092

"abc$" in ""

matchhere / matchhere\_iter = 1330000 / 1310000 = 1.0153

"abc$" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 323529 / 390000 = 0.8296

"abc$" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 1100000 / 1100000 = 1.0000

"a\*b\*c\*d" in ""

matchhere / matchhere\_iter = 274510 / 277228 = 0.9902

"a\*b\*c\*d" in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 94340 / 95238 = 0.9906

"a\*b\*c\*d" in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 223301 / 223301 = 1.0000

".................." in ""

matchhere / matchhere\_iter = 1340000 / 1310000 = 1.0229

".................." in "abc%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 90909 / 120370 = 0.7552

".................." in "%%%%%%%%%%%%%%%%%%%%%abc%%%%%%%%%%%%%%%%%%%%%"

matchhere / matchhere\_iter = 90909 / 120370 = 0.7552

The non-recursive version is faster when it must read over many pattern

characters. See the 20% improvement matching "abc" against "abc%%%..."

and the 32% improvement in the contrived pattern consisting only of

dots. Performance with other patterns was about the same.

Exercise 9-8

See grep.c. I added the -n, -v, and -c (count only) options. I omitted

the -i option because I'm looking forward to exercise 9-13 and I know

that case folding in Unicode is complex; I don't want that to get in the

way of UTF-8 support. I print line numbers the same way GNU grep does:

on the same line as the text, after the file name.

Exercise 9-9

See match.c.

Exercise 9-10

See match.c and the compile function. I decided to make the leap to a

compiled form of regular expressions to make this and upcoming exercises

easier.

Exercise 9-11

See match.c, charclass.c, and charclass.h. I might have used bit vectors

to represent character classes, but seeing exercise 9-13 coming I

decided to use sorted linked lists of character ranges.

Exercise 9-12.

See match.c.

Exercise 9-13.

See match.c, utf8.c, and utf8.h. I read the Thompson and Pike UTF-8

paper (http://www.cl.cam.ac.uk/~mgk25/ucs/UTF-8-Plan9-paper.pdf). From

there I got some implementation ideas and the name "Rune" for a Unicode

character.

The primary UTF-8-handling function is utf8\_to\_rune, which decodes UTF-8

bytes into a single character. The inverse function rune\_to\_utf8 is not

needed by grep but I used it in the utf8cat program. utf8cat reads a

UTF-8 file and prints it out again, replacing any invalid sequences with

U+FFFD.

For testing I used these files:

http://www.cl.cam.ac.uk/~mgk25/ucs/UTF-8-demo.txt

http://www.cl.cam.ac.uk/~mgk25/ucs/UTF-8-test.txt

Test grep with a command like

./grep $'.\xce\xba[a-\xe1\xbd\xb9]\xcf\x83?\xce\xbc\xce\xb5!\*' \

UTF-8-demo.txt UTF-8-test.txt

That searches for the Greek "kosme" in both files, while exercising some

regular expression features. Or

./grep ' .homoiousian. ' mission.stmt

to see that the multibyte quote characters are recognized as single

characters.

grep.c and gres.c required no changes to work with UTF-8 once the

functions in match.c were changed.

Exercise 9-14

See re-test.py, rematch.c, and rematch.py. Python has built-in support

for Unicode and UTF-8 so it is not hard to build and encode the regular

expressions and test strings. Unfortunately the build of Python I used

was compiled with two-byte wide characters, so the tester is limited to

characters whose value is 0xFFFF or under.

The tester found a few bugs in my implementation: Invalid UTF-8

sequences were converted to UTF8\_BADCHAR and stored in the compiled

regular expression, where they could match invalid characters in the

matched string. Backslash escapes were not recognized within character

classes.

Bigger than the bugs, however, were the changes I had to make so the

regular expression engine would work the way I wanted. I had the idea to

compile invalid characters in the regular expression to special SLUG

atoms that would not match any string. But then I didn't know what to do

with expressions like '[a-<XX>]', where <XX> is an invalid UTF-8

sequence. I first compiled the whole character class into a slug, but

that creates a contradiction with '[^a-<XX>]'; two expressions that

should be complementary now mean the same thing. I decided to reject

with an error all expressions containing an invalid character, and this

required some architectural changes. I continued to allow invalid

characters in input strings, but they do not match anything, not even

'.'. So 'a<XX>c' matches the expressions '^a' and 'cd\*', but not '...'.

I also tested the re module in Python 2.5. rematch.py is the interface

to it. The only tests it failed were those that dealt with invalid

characters, the correct treatment of which is subject to interpretation.

I chose to report an error if an invalid character appeared in an

expression, and not to allow an invalid character in a search string to

match anything in an expression. Python allows invalid characters in

either place. Hence all the test failures were of this sort:

match(u'.', u'\ud8d7'): expected NOMATCH, got MATCH.

match(u'[\x01-\ufffd]', u'\ud8d7'): expected NOMATCH, got MATCH.

match(u'\ud8d7', u'\ud8d6'): expected ERROR, got NOMATCH.

match(u'\ud8d7', u'\ud8d7'): expected ERROR, got MATCH.

Exercise 9-15

See quine.c, quine.py, and quine.rb. I decided to first try to write the

program in Ruby, a language I had never used. My first Ruby program used

an array of integers representing character values, like the C version.

I was able to refine it to this shorter form. The C version is

conceptually the same but longer because of the necessary boilerplate

and the lack of string-building functions. I wrote the Python version

last. It is pretty much identical to the Ruby one.

Exercise 9-16

See op-cost-c++.cpp. I originally wrote the code with macros for

exercise 7-7.

Exercise 9-17

See OpCost.java.in. This is how I originally wrote the program for

exercise 7-8. The C preprocessor writes the Java program.

Exercise 9-18

A good way would be to compile the entire expression once, then execute

it to compute the value. The compiler could then go back and write code

to load the value over the top of the compiled expression. Or, if the

compiler also has an interpreter, such that not all expressions are

compiled but only commonly used ones, then it would be convenient to

have the interpreter compute the value directly without the overhead of

compilation.

Exercise 9-19

Much of the testing would be the same as for any other compiler or

interpreter. It would involve compiling short test programs and tricky

expressions, then running them and checking for the expected output. A

special difficulty with the on-the-fly compiler would be checking that

all the memory manipulation is safe, that compiled code doesn't cause

the program counter to fly off into uncharted memory. I think that

fixing these problems would account for much of the debugging. For this

my first instinct would be to run the programs inside a protected

environment, such as within Valgrind, to check that there are no

incorrect memory accesses. Another way would be to create checked

buffers around the compiled memory zones in the manner of exercise 5-1.