COMSM1201 : Exercises in C

Neill Campbell

Department of Computer Science, University of Bristol

Copyright © 2019 Neill Campbell

Formatted in $\LaTeX$, based on the Legrand Orange Book from BOOK-WEBSITE.COM

This work is licensed under a Creative Commons Attribution-NonCommerical-ShareAlike 4.0 International License.

Contents

2. Pre-Arrays

Note that these exercises are in **NO** particular order - try the ones you find easy before attempting more complex ones.

2.1 Planet Trium

On the planet Trium, everyone's name has three letters. Not only this, all the names take the form of non-vowel, vowel, non-vowel.

Exercise 2.1 Write a program that outputs all the valid names and numbers them. The first few should look like :

```
 1 bab
 2 bac
 3 bad
 4 baf
 5 bag
 6 bah
 7 baj
 8 bak
 9 bal
10 bam
11 ban
12 bap
13 baq
14 bar
15 bas
16 bat
17 bav
18 baw
19 bax
20 bay
21 baz
```

```
22 beb
23 bec
24 bed
25 bef
26 beg
```

■

2.2 Planet Bob

On the planet Bob, everyone's name has three letters. These names either take the form of consonant-vowel-consonant or else vowel-consonant-vowel. For the purposes here, vowels are the letters `{a,e,i,o,u}` and consonants are all other letters. There are two other rules :

1. The first letter and third letters of the name must always be the same.
2. The name is only 'valid' if, when you sum up the values of the three letters ($a = 1, b = 2$ etc.), the sum is prime.

The name "bob" is a valid name: it has the form consonant-vowel-consonant, the first letter and third letters are the same ('b') and the three letters sum to $19(2+15+2)$, which is prime. The name "aba' is **not** valid, since the sum of the three letters is $4(1+2+1)$ which is **not** prime.

Exercise 2.2 Write a program that outputs all the valid names and numbers them. The first few names should look like :

```
 1 aca
 2 aka
 3 aqa
 4 bab
 5 bib
 6 bob
 7 cac
 8 cec
 9 ded
10 did
11 dod
12 dud
13 ece
14 ege
15 eme
16 ese
17 faf
```

■

2.3 Secret Codes

Write a program that converts a stream of text typed by the user into a 'secret' code. This is achieved by turning every letter 'a' into a 'z', every letter 'b' into a 'y', every letter 'c' into and 'x' and so on.

Exercise 2.3 Write a function whose 'top-line' is :

```
int scode(int a)
```

that takes a character, and returns the secret code for this character. Note that the function

does need to preserve the case of the letter, and that non-letters are returned unaffected.

When the program is run, the following input:

```
The Quick Brown Fox Jumps Over the Lazy Dog !
```

produces the following output :

```
Gsv Jfrxp Yildm Ulc Qfnkh Levi gsv Ozab Wlt !
```

■

2.4 Cash Machine (ATM)

Some cash dispensers only contain £20 notes. When a user types in how much money they'd like to be given, you need to check that the amount requested can be dispensed exactly using only £20 notes. If not, a choice of the two closest (one lower, one higher) amounts is presented.

Exercise 2.4 Write a program that inputs a number from the user and then prompts them for a better choice if it is not correct. For example :

```
How much money would you like ? 175
I can give you 160 or 180, try again.
How much money would you like ? 180
OK, dispensing ...
```

or :

```
How much money would you like ? 25
I can give you 20 or 40, try again.
How much money would you like ? 45
I can give you 40 or 60, try again.
How much money would you like ? 80
OK, dispensing ...
```

In this assessment you may assume the input from the user is "sensible" i.e. is not a negative number etc. ■

2.5 Hailstone Sequence

Hailstones sequences are ones that seem to always return to 1. The number is halved if even, and if odd then the next becomes `3*n+1`. For instance, when we start with the number 6, we get the sequence : `6, 3, 10, 5, 16, 8, 4, 2, 1` that has nine numbers in it. When we start with the number 11, the sequence is longer, containing 15 numbers : `11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1`.

Exercise 2.5 Write a program that :

- displays which initial number (less than 50,000) creates the **longest** hailstone sequence.
- displays which initial number (less than 50,000) leads to the **largest** number appearing in the sequence.

■

2.6 Monte Carlo Π

At :

http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html

a square whose sides are of length r, and a quarter-circle, whose radius is of r are drawn.

If you throw random darts at the square, then many, but not all, also hit the circle. A dart landing at position (x,y) only hits the circle if $x^2+y^2 \le r^2$.

The area of the circle is $\frac{\pi}{4}r^2$, and the area of the square is r^2.

Therefore, a way to approximate π, is to choose random (x,y) pairs inside the square h_a, and count the h_c ones that hit the circle. Then:

$$\pi \approx \frac{4h_c}{h_a} \tag{2.1}$$

Exercise 2.6 Write a program to run this simulation, and display the improving version of the approximation to π. ■

2.7 Leibniz Π

See:

https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80

The Mathematical constant π can be approximated using the formula :

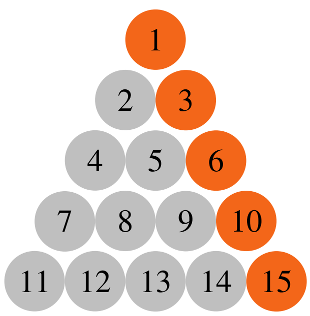
$$\pi = 4 - \tfrac{4}{3} + \tfrac{4}{5} - \tfrac{4}{7} + \tfrac{4}{9} - \tfrac{4}{11} + \dots$$

Notice the pattern here of alternating $+$ and $-$ signs, and the odd divisors.

Exercise 2.7 Write a program that computes π looping through smaller and smaller fractions of the series above. How many iterations does it take to get π correctly approximated to 9 digits ? ■

2.8 Triangle Numbers

A Triangle number is the sum of numbers from 1 to n. The 5^{th} Triangle number is the sum of numbers $1,2,3,4,5$, that is 15. They also relate to the number of circles you could stack up as equilateral triangles



$T_1 = 1$

$T_2 = 3$

$T_3 = 6$

$T_4 = 10$

$T_5 = 15$

Exercise 2.8 Write a program that prints out the sequence of Triangle numbers, using iteration, computing the next number based upon the previous.

Check these against:

www `http://oeis.org/A000217`

and also by generating the n^{th} Triangle number based on the Equation :

$$T_n = n * (n+1)/2$$

2.9 Higher-Lower

In the game "higher-Lower", a user has to guess a secret number chosen by another. They then repeatedly guess the number, being only told whether their guess was greater, or less than the secret one.

Exercise 2.9 Write a program that selects a random number between 1 and 1000. The user is asked to guess this number. If this guess is correct, the user is told that they have chosen the correct number and the game ends. Otherwise, they are told if their guess was too high or too low. The user has 10 goes to guess correctly before the game ends and they lose.

3. 1D Arrays and Strings

For some of these exercises you'll need to understand command line input to `main()` from the shell, often simply referred to as `argc/argv` in C. Please see:

www `http://www.thegeekstuff.com/2013/01/c-argc-argv/`

for more information about this.

3.1 Neill's Microwave

Last week I purchased a new, state-of-the-art microwave oven. To select how long you wish to cook food for, there are three buttons: one marked "10 minutes", one marked "1 minute" and one marked "10 seconds". To cook something for 90 seconds requires you to press the "1 minute" button, and the "10 seconds" button three times. This is four button presses in total. To cook something for 25 seconds requires three button presses; the "10 second" button needs to be pressed three times and we have to accept a minor overcooking of the food.

Exercise 3.1 Using an array to store the cooking times for the buttons, write a program that, given a required cooking time in seconds, allows the minimum number of button presses to be determined.

Example executions of the program will look like :

```
Type the time required
25
Number of button presses = 3
Type the time required
705
Number of button presses = 7
```

■

3.2 Music Playlisters

Most MP3 players have a "random" or "shuffle" feature. The problem with these is that they can sometimes be **too** random; a particular song could be played twice in succession if the new song to play is truly chosen randomly each time without taking into account what has already been played.

To solve this, many of them randomly order the entire playlist so that each song appears in a random place, but once only. The output might look something this:

```
How many songs required ? 5
4 3 5 1 2
```

or :

```
How many songs required ? 10
1 9 10 2 4 7 3 6 5 8
```

Exercise 3.2 Write a program that gets a number from the user (to represent the number of songs required) and outputs a randomised list.

Exercise 3.3 Rewrite Exercise 3.2 so that the program passes an array of integers (e.g. `[1,2,3,4,5,6,7,8,9,10]`) to a function and re-orders them **in-place** (no other arrays are used) and with an algorithm having complexity $O(n)$.

3.3 Rule 110

Rather interesting patterns can be created using *Cellular Automata*. Here we will use a simple example, one known as *Rule 110* : The idea is that in a 1D array, cells can be either on ■ or off □ (perhaps represented by the integer values 1 and 0). A new 1D array is created in which we decide upon the state of each cell in the array based on the cell above and its two immediate neighbours.

If the three cells above are all 'on', then the cell is set to 'off' ($111 \to 0$). If the three cells above are 'on', 'on', 'off' then the new cell is set to 'on' ($110 \to 1$). The rules, in full, are:
$111 \to 0$
$110 \to 1$
$101 \to 1$
$100 \to 0$
$011 \to 1$
$010 \to 1$
$001 \to 1$
$000 \to 0$

You take a 1D array, filled with zeroes or ones, and based on these, you create a new 1D array of zeroes and ones. Any particular cell uses the three cells 'above' it to make the decision about its value. If the first line has all zeroes and a single one in the middle, then the automata evolves as:

Exercise 3.4 Write a program that outputs something similar to the above using plain text, giving the user the option to start with a randomised first line, or a line with a single 'on' in the central location.

Exercise 3.5 Rewrite the program above to allow other rules to be displayed - for instance $124, 30$ and 90.

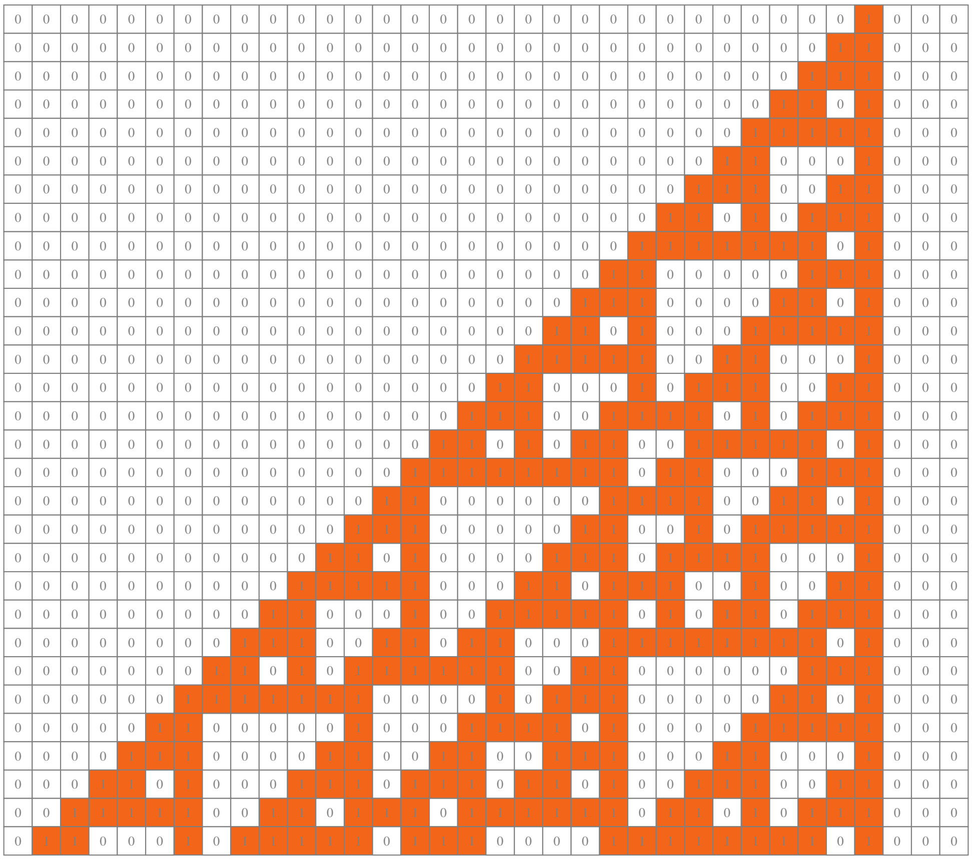


Figure 3.1: 1D cellular automaton using Rule 110. Top line shows initial state, each subsequent line is produced from the line above it. Each cell has a rule to switch it ‘on’ or ’off’ based on the state of the three cells above it in the diagram.

www

`http://en.wikipedia.org/wiki/Rule_110`

3.4 Palindromes

From `wikipedia.org` :

> A palindrome is a word, phrase, number or other sequence of units that has the property of reading the same in either direction (the adjustment of punctuation and spaces between words is generally permitted).
>
> The most familiar palindromes, in English at least, are character-by-character: the written characters read the same backwards as forwards. Palindromes may consist of a single word (such as "civic" or "level"), a phrase or sentence ("Neil, a trap! Sid is part alien!", "Was it a rat I saw?") or a longer passage of text ("Sit on a potato pan, Otis."), even a fragmented sentence ("A man, a plan, a canal: Panama!", "No Roman a moron"). Spaces, punctuation and case are usually ignored, even in terms of abbreviation ("Mr. Owl ate my metal worm").

Exercise 3.6 Write a program that prompts a user for a phrase and tells them whether it is a palindrome or not. **Do not** use any of the built-in string-handling functions (`string.h`), such as `strlen()` and `strcmp()`. However, you **may** use the character functions (`ctype.h`), such as `islower()` and `isalpha()`.

Check you program with the following palindromes :

```
"kayak"
"A man, a plan, a canal: Panama!"
"Madam, in Eden I'm Adam,"
"Level, madam, level!"
```

3.5 Int to String

Exercise 3.7 Write a function that converts an integer to a string, so that the following code snippet works correctly:

```
int i;
char s[256];
scanf("%d", &i);
int2string(i,s);
printf("%s\n", s);
```

The integer may be signed (i.e. be positive or negative) and you may assume it is in base-10.

Avoid using any of the built-in string-handling functions to do this (e.g. `itoa()`!) including those in `string.h`.

3.6 Roman Numerals

Adapted from:

WWW `http://mathworld.wolfram.com/RomanNumerals.html`

> "Roman numerals are a system of numerical notations used by the Romans. They are an additive (and subtractive) system in which letters are used to denote certain "base" numbers, and arbitrary numbers are then denoted using combinations of symbols. Unfortunately, little is known about the origin of the Roman numeral system.
>
> The following table gives the Latin letters used in Roman numerals and the corresponding numerical values they represent :

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

> For example, the number 1732 would be denoted MDCCXXXII in Roman numerals. However, Roman numerals are not a purely additive number system. In particular,

instead of using four symbols to represent a 4, 40, 9, 90, etc. (i.e., IIII, XXXX, VIIII, LXXXX, etc.), such numbers are instead denoted by preceding the symbol for 5, 50, 10, 100, etc., with a symbol indicating subtraction. For example, 4 is denoted IV, 9 as IX, 40 as XL, etc."

It turns out that every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

I	1	IV	4
V	5	IX	9
X	10	XL	40
L	50	XC	90
C	100	CD	400
D	500	CM	900
M	1000		

Exercise 3.8 Write a program that reads a roman numeral (in the range `1 - 3999`) and outputs the corresponding valid arabic integer. Amongst others, check that `MCMXCIX` returns `1999`, `MCMLXVII` returns `1967` and that `MCDXCI` returns `1491`.

You should use the following template :

```
#include <stdio.h>

int romanToArabic( char *roman );

int main(int argc, char **argv)
{
    if( argc==2 ){
        printf("The roman numeral %s is equal to %d\n", \
        argv[1], romanToArabic(argv[1]));
    }else{
        printf("ERROR: Incorrect usage, try e.g. %s XXI\n", argv[0]);
    }
    return 0;
}
```

You need to add the function `romanToArabic()`. ■

3.7 Soundex Coding

First applied to the 1880 census, Soundex is a phonetic index, not a strictly alphabetical one. Its key feature is that it codes surnames (last names) based on the way a name sounds rather than on how it is spelled. For example, surnames that sound the same but are spelled differently, like Smith and Smyth, have the same code and are indexed together. The intent was to help researchers find a surname quickly even though it may have received different spellings. If a name like Cook, though, is spelled Koch or Faust is Phaust, a search for a different set of Soundex codes and cards based on the variation of the surname's first letter is necessary.

To use Soundex, researchers must first code the surname of the person or family in which they are interested. Every Soundex code consists of a letter and three numbers, such as B536, representing names such as Bender. The letter is always the first letter of the surname, whether it is a vowel or a consonant.

The detailed description of the algorithm may be found at :

www http://www.highprogrammer.com/alan/numbers/soundex.html

The first letter is simply the first letter in the word. The remaining numbers range from 1 to 6, indicating different categories of sounds created by consanants following the first letter. If the word is too short to generate 3 numbers, 0 is added as needed. If the generated code is longer than 3 numbers, the extra are thrown away.

Code	*Letters Description*
1	*B, F, P, V Labial*
2	*C, G, J, K, Q, S, X, Z Gutterals and sibilants*
3	*D, T Dental*
4	*L Long liquid*
5	*M, N Nasal*
6	*R Short liquid*
SKIP	*A, E, H, I, O, U, W, Y Vowels (and H, W, and Y) are skipped*

There are several special cases when calculating a soundex code:

- *Letters with the same soundex number that are immediately next to each other are discarded. So Pfizer becomes Pizer, Sack becomes Sac, Czar becomes Car, Collins becomes Colins, and Mroczak becomes Mrocak.*
- *If two letters with the same soundex number seperated by "H" or "W", only use the first letter. So Ashcroft is treated as Ashroft.*

Sample Soundex codes:

Word	*Soundex*
Washington	*W252*
Wu	*W000*
DeSmet	*D253*
Gutierrez	*G362*
Pfister	*P236*
Jackson	*J250*
Tymczak	*T522*
Ashcraft	*A261*

Exercise 3.9 Write a program that takes the name entered as `argv[1]` and prints the corresponding soundex code for it. ■

4. 2D Arrays

4.1 The Game of Life

The Game of Life was developed by British mathematician John Horton Conway. In Life, a board represents the world and each cell a single location. A cell may be either empty or inhabited. The game has three simple rules, which relate to the cell's eight nearest neigbours :

1. **Survival** An inhabited cell remains inhabited if exactly 2 or 3 of its neighbouring cells are inhabited.
2. **Death** An inhabited cell becomes uninhabited if fewer than 2, or more than 3 of its neighbours are inhabited.
3. **Birth** An uninhabited cell becomes inhabited if exactly 3 of its neighbours are inhabited.

The next board is derived solely from the current one. The current board remains unchanged while computing the next board. In the simple case shown here, the boards alternate infinitely between these two states.

0	0	0	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

The 1.06 format

A general purpose way of encoding the input board is called the Life 1.06 format :

www `http://conwaylife.com/wiki/Life_1.06`

This format has comments indicated by a hash in the first column, and the first line is always:

```
#Life 1.06
```

Every line specifies an x and y coordinate of a live cell; such files can be quite long. The coordinates specified are relative to the middle of the board, so 01 means the middle row, one cell to the right of the centre.

There are hundreds of interesting patterns stored like this on the above site.

Exercise 4.1 Write a program which is run using the `argc` and `argv` parameters to `main`. The usage is as follows :

```
% life file1.lif 10
```

where `file1.lif` is a file specifying the inital state of the board, and `10` specifys that ten iterations are required.

Display the output to screen every iteration using plain text, you may assume that the board is 150 cells wide and 90 cells tall. ■

Alternative Rules for The Game of Life

The rules for life could also be phrased in a different manner, that is, give birth if there are two neighbours around an empty cell (B2) and allow an 'alive' cell to survive only if surrounded by 2 or 3 cells (S23). Other rules which are *life-like* exist, for instance *34 Life* (B34/S34), *Life Without Death* (B3/S012345678) and HighLife (B36/S23).

www `http://en.wikipedia.org/wiki/Life-like_cellular_automaton`

Exercise 4.2 Write a program that allows the user to input life-like rules e.g. :

```
life B34/S34 lifeboard.lif
```

or

```
life B2/S lifeboard.lif
```

and display generations of boards, beginning with the inital board in the input file. ■

4.2 Life Wars

Inspired by the classic game, *Core Wars*

www `http://en.wikipedia.org/wiki/Core_War`

here we look at a two player version of Conway's Game of Life.

In our game, each of two 'players' submit a Life 1.06 file and cells from these inserted into an empty board. The cells are coded based on which player created them (say '+' and '@'). The game is then run, and the player having most cells left after a fixed number of iterations, over many games, is deemed the winner.

The rules are :

1. The board is 150 cells wide, and 90 cells tall.
2. The board is toroidal; that is, it wraps around from the left edge to the right, and the top to the bottom.
3. Each player can submit a Life 1.06 file that is maximum of 100 lines long, **including** the header line.
4. Since each of the Life 1.06 files describe absolute positions for cells, each player is assigned a random origin for their cells. If there is any collision when attempting to add the cells initially (i.e. both players happen to specify a cell in the same square), then new origins are chosen randomly and the process begun from scratch.

5. The ‘standard’ B3/S23 rules are used.
6. The colour of cells is never taken into account (i.e. cells are still either ‘alive’ or ‘dead’). The sole exception to this is that when a cell is born, it takes the colour of the majority its neighbours.
7. There are 5000 generations played every game.
8. A running count of how many cells each player has left at the end of each game is kept. The board is cleared and the next game randomly restarted.
9. There are 50 games run in total.
10. The player with the highest number of cells over all 50 games is the winner.

It’s easy to extend these rules, of course, to allow for three or more players, but it’s unclear for three players what would happen in the majority vote if there was a draw (i.e. a new cell is born based on three cells around it, one belonging to each player. Other rule variants are also possible (e.g. *Highlife* (B36/S23), but once again, the birth rule for 3 or 6 neighbours would cause majority voting issues for two and three players.

Exercise 4.3 Write a program that accepts two Life 1.06 files and reports which of them is the winner. The first few games might look like:

```
% /lifewars blinkerpuffer2.lif bigglider.lif

        0    50    12   Player 1
        1   370   141   Player 1
        2   437   281   Player 1
        3   450   602   Player 2
        4   540   623   Player 2
        5   991   629   Player 1
        6  1063   674   Player 1
        7  1211   707   Player 1
        8  1263   735   Player 1
        9  1358   758   Player 1
.
.
.
    Player 1 wins by 7857 cells to 2373 cells
```

■

4.3 Wireworld

Wireworld is a cellular automaton due to Brian Silverman, formed from a 2D grid of cells, designed to simulate digital electronics. Each cell can be in one of four states, either ‘empty’, ‘electron head’, ‘electron tail’ or ‘copper’ (or ‘conductor’).

The next generation of the cells follows the rules, where n is the number of electron heads found in the 8-surrounding cells:

- empty $->$ empty
- electron head $->$ electron tail
- electron tail $->$ copper
- copper $->$ electron head if $n == 1$ or $n == 2$
- copper $->$ copper otherwise

See also:

www https://en.wikipedia.org/wiki/Wireworld

www http://www.heise.ws/fourticklogic.html

Exercise 4.4 Write a program which is run using the `argc` and `argv` parameters to `main`. The usage is as follows :

```
$ wireworld wirefile.txt
```

where `wirefile.txt` is a file specifying the initial state of the board. This file codes empty cells as ‘ ’, heads as ‘H’, tails as ‘t’ and copper as ‘c’. Display the board for 1000 generations using plain text. You may assume that the grid is always 40 cells by 40

Make sure all your code is fully ANSI compliant, and fully follows the house-style guidelines. Show that your code has been developed using short, well-tested functions via the use of `assert()` testing.

4.4 ncurses

C has no inherent functionality to allow printing in colour etc. Therefore, a programming library know a `ncurses` was created in 1993 to allow terminals to interpret certain control-codes as colours and other effects.

The library itself is somewhat complex, allowing keyboard and mouse events to be captured and a whole range of simple graphics functionality. On the web page is my ‘wrapper’ for the library, along with a program demonstrating its use. This will only work in unix-style terminals. Note that after you begin ncurses mode (using `Neill_NCURS_Init()`) that you can’t print to stdout or stderr, until you switch it off (using `Neill_NCURS_Done()`).

To compile the code you’ll have to use both my code `neillncurses.c` and also link in the ncurses library. A typical compile might look like

```
gcc yourcode.c neillncurses.c -Wall -Wfloat-equal -Wextra -O2
    -pedantic -ansi -lncurses -lm
```

If you’re running a virtual box you may also need to install the `ncurses` developer files, including `ncurses.h`, using:

```
sudo apt install libncurses-dev
```

Some terminals do not support ncurses, so make sure you are using an ‘xterm’ or equivalent.

Exercise 4.5 Adapt the wireworld code in Exercise 4.4 so that the output is displayed using this library, with tails being red, heads being blue, copper being yellow and background being black. The main loop will update the board, display it, and repeat until a quit event occurs (e.g. a mouse click or the ESC key is pressed).

Exercise 4.6 Adapt the life code in Exercise 4.1 so that the output is displayed using this library, with sensible choices made for cell colours. The main loop will update the board, display it, and repeat until a quit event occurs (e.g. a mouse click or the ESC key is pressed). ■

5. Strings, Recursion and SDL

5.1 Anagrams

An anagram is a collection of letters that when unscrambled, using all the leters, make a single word. For instance `magrana` can be rearranged to make the word `anagram`.

Exercise 5.1 Using a file of valid words, allow the user to enter an anagram, and have the answer(s) printed. For instance :

```
 % ./anagram sternaig
angriest
astringe
ganister
gantries
ingrates
rangiest
reasting
stearing
```

■

Exercise 5.2 Using a file of valid words, find all words which are anagrams of each other. Each word should appear in a maximum of one list. Output will look something like :

```
 % ./selfanagram
.
.
7 merits mister miters mitres remits smiter timers
.
.
.
6 noters stoner tenors tensor toners trones
.
.
.
6 opts post pots spot stop tops
```

```
.
.
.
6 restrain retrains strainer terrains trainers transire
.
```

If you wished to create "interesting" anagrams, rather than simply a random jumble of letters, you could combine together two shorter words which are an anagram of a longer one.

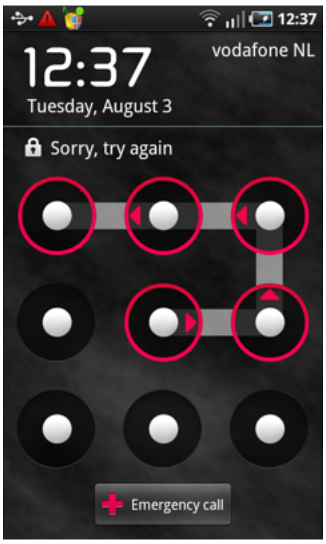
Exercise 5.3 Write a program which uses an exhaustive search of all the possible pairs of short words to make the target word to be computed. For instance, a few of the many pairs that can be used to make an anagram of `compiler` are :

```
 % ./teabreak compiler
LiceRomp
LimeCrop
LimpCore
MileCrop
MoreClip
PermCoil
PromLice
RelicMop
```

The name `Campbell` comes out as `CalmPleb` which is a bit harsh. Can't **ever** remember being called calm ...

5.2 Draw to Unlock

Rather than remembering passwords or passcodes, many mobile devices now allow the user to draw a pattern on the screen to unlock them.



Here we will explore how many unique patterns are available when drawing such patterns to connect "dots", such as shown in the figure. We assume that people put their finger on one "dot" and then only ever move one position left, right, up or down (but never diagonally) at a time.

You are not allowed to return to a “dot” once it has been visited once. If we number the first position in our path as 1, the second as 2 and so on, then beginning in the top left-hand corner, some of the possible patterns of 9 moves are :

```
1 2 3      1 2 3      1 2 3
6 5 4      8 9 4      8 7 4
7 8 9      7 6 5      9 6 5
```

Exercise 5.4 Write a program that computes and outputs all the valid paths. Use **recursion** to achieve this.

- How many different patterns of length 9 are available on a 3×3 grid, if the user begins in the top left corner ?
- How many different patterns of length 9 are available on a 3×3 grid, if the user begins in the middle left ?
- How many different patterns of length 7 are available on a 3×3 grid, if the user begins in the top left corner ?
- How many different patterns of length 25 are available on a 5×5 grid, if the user begins in the top left corner ?

■

5.3 SDL - Intro

Many programming languages have no inherent graphics capabilities. To get windows to appear on the screen, or to draw lines and shapes, you need to make use of an external library. Here we use SDL[1], a cross-platform library providing the user with (amongst other things) such graphical capabilities.

WWW `https://www.libsdl.org/`

The use of SDL is, unsurprisingly, non-trival, so some simple wrapper files have been created (`neillsdl2.c` and `neillsdl2.h`). These give you some simple functions to initialise a window, draw rectangles, wait for the user to press a key etc.

An example program using this functionality is provided in a file `blocks.c`.

This program initialises a window, then sits in a loop, drawing randomly positioned and coloured squares, until the user presses the mouse or a key.

Exercise 5.5 Using the `Makefile` provided, compile and run this program, using something like : `make -f sdl_makefile`

SDL is already installed on lab machines. At home, if you’re using a ubuntu-style linux machine, use: `sudo apt install libsdl2-dev` to install it. ■

5.4 Word Ladders

In this game, made famous by the author Lewis Caroll, and investigated by many Computer Scientists including Donald Knuth, you find missing words to complete a sequence. For instance, you might be asked how go from “WILD” to “TAME” by only changing one character at a time:

[1] actually, we are using the most recent version SDL2, which is installed on all the lab machines

W I L D
W I L E
T I L E
T A L E
T A M E

A useful concept here is that of the `edit distance`. Here, the edit distance is a count of the number of characters which are different between two words. For words of n characters, the edit distance will be in the range $0 \dots n$. For 'aboard' and 'canape' the edit distance is 5. An edit distance of zero means that the words are identical.

In a heavily constrained version of this game we make some simplifying assumptions:

- Words are always four letters long.
- We only seek ladders of five words in total.
- Only one letter is changed at a time.
- A letter is only changed from its initial state, to its target state. This is important, since if you decide to change the second letter then you will always know what it's changing from, to what it's changing to.

So, in the example above, it is enough to give the first word, the last word, and the position of the character which changed on each line. On line one, the fourth letter 'D' was changed to an 'E', on the next line the first character 'W' was changed to a 'T' and so on. The whole ladder can be defined by "WILD", "TAME" and the sequence $4, 1, 2, 3$.

W I L D
W I L E
T I L E
T A L E
T A M E

Since each letter changes exactly once, the order in which this happens is a *permutation* of the numbers $1, 2, 3, 4$, which we have looked at elsewhere..

We'll also need another function:

```
int edit_distance(char *s, char *t);
```

which returns the number of characters which are different between two strings of the same length. For our strings of length four (excluding the null character) this will be either $0, 1, 2, 3$ or 4. Here, we can check that the first and last word in our search are distance 4 apart.

Exercise 5.6 For the constrained version of the game, given a file of valid four letter words, write a program which when given two words on the command line (`argv[1]` and `argv[2]`) outputs the correct solution, if available. Use an exhaustive search over all 24 permutations until one leads to no invalid words being required. Make sure your program works, with amongst others, the following:

C O L D	P O K E	C U B E
C O R D	P O L E	C U B S
C A R D	P O L L	T U B S
W A R D	M O L L	T U N S
W A R M	M A L L	T O N S

Exercise 5.7 Adapt the program above so that if the first and last words share a letter (the edit distance is less than 4), you can find the word ladder required, as in:

W A S P
W A S H
W I S H
F I S H

For the "full" version of Wordladder, you make no assumptions about the number of words that are needed to make the ladder, although we do assume that all the words in the ladder are the same size.

Exercise 5.8 To achieve this, you could make a list of all the words, and for all words an `edit distance` of 1 away from the initial word, mark these and store their 'parent'. Now, go through this list, and for all words marked, find words which are distance 1 from these, and hence distance 2 from the initial word. Mark these and retain their parent. Be careful you don't use words already marked. If the word ladder is possible, you'll eventually find the solution, and via the record of the parents, have the correct route. This is shown for the word ladder CAT to DOG in Figure 5.1 using a very small subset of the possible three letter words.

5.5 The Devil's Dartboard

In the traditional 'pub' game, darts, there are 62 different possible scores : single 1 - 20 (the white and black areas), double 1 - 20 (the outer red and green segments) (i.e. 2, 4, 6, 8 ...), treble 1 - 20 (i.e. 3, 6, 9, 12 ...) (the inner red or green segments), 25 (small green circle) and 50 (the small red inner circle).

It's not obvious, if you were inventing darts from scratch, how best to lay out the numbers. The London board shown seems to have small numbers near high numbers, so that if you just miss the 20 for example, you'll hit a small number instead.

Here we look at a measure for the 'difficulty' of a dartboard. One approach is to simply sum up the values of adjacent triples, squaring this number. So for the London board shown, this would be: $(20+1+18)^2+(1+18+4)^2+(18+4+13)^2\ldots(5+20+1)^2=20478$

For our purposes a **lower** number is better[2]. For more details see :

WWW `http://www.mathpages.com/home/kmath025.htm`

Exercise 5.9 Write a program that repeatedly chooses two positions on the board and swaps their numbers. If this leads to a lower cost, keep the board. If not, unswap them. Repeat this *greedy search* 5000000 times, and print out the best board found. Begin with the trivial monotonic sequence. The output may look something like :

```
Total = 19966 :  3 19 11  2 18 12  1 20 10  4 16  8
```

[2] It's beyond the scope here to explain why!

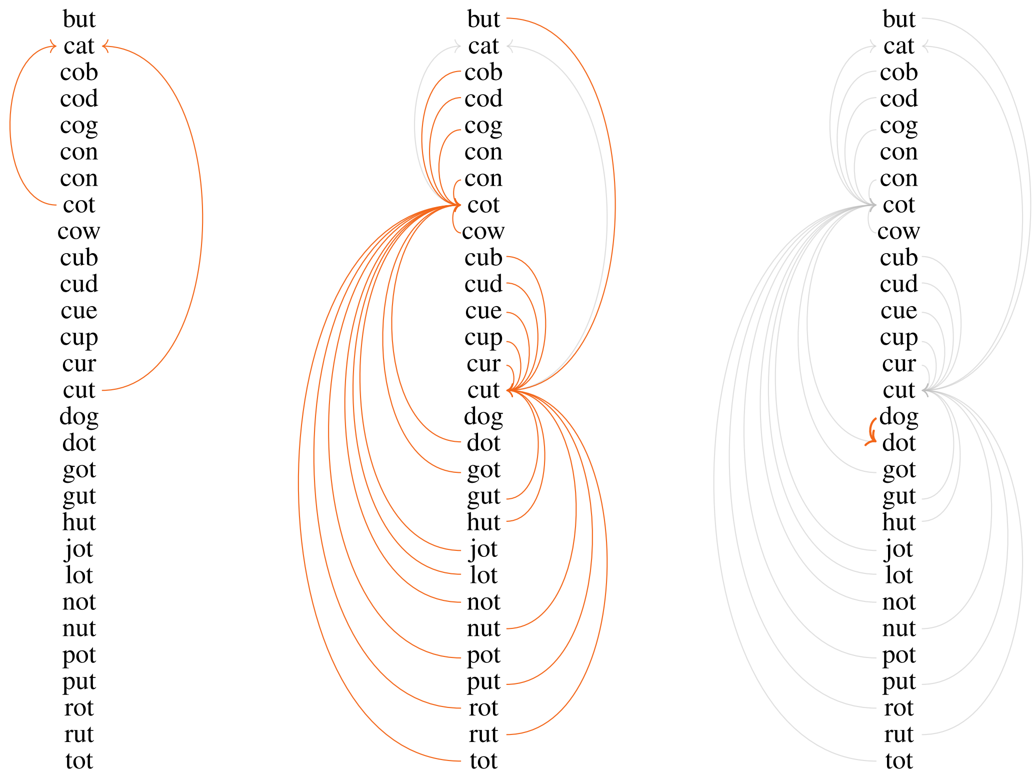
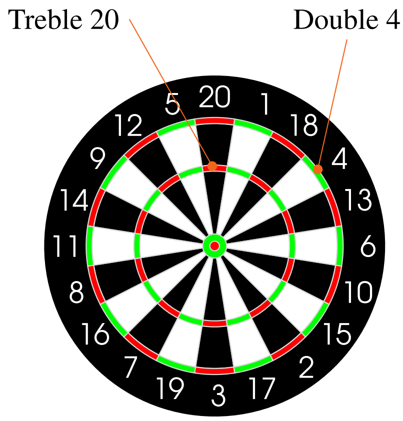


Figure 5.1: Word Ladder from CAT to DOG. (Left) Words which are distance one from CAT are COT and CUT. (Middle) Words which are distance one from CUT and COT, which includes amongst others, DOT. (Right) DOG is distance one from DOT. We now have our route, via the pointers, back to CAT.




```
14  5 13 15  6  7 17  9
```

or

```
Total = 19910 :  3 18 10  5 16  9  8 14 11  4 19  6
7 20  2 13 15  1 17 12
```

The score of 19874 seems to be the lowest possible that may be obtained via this technique.

■

5.6 Maze

Escaping from a maze can be done in several ways (ink-blotting, righthand-on-wall etc.) but here we look at recursion.

Exercise 5.10 Write a program to read in a maze typed by a user via the filename passed to `argv[1]`. You can assume the maze will be no larger than 20×20, walls are designated by with a # and the rest are spaces. The entrance can be assumed to be the gap in the wall closest to (but not necessarilty exactly at) the top lefthand corner. The sizes of the maze are given on the first line of the file (width,height). Write a program that finds the route through a maze, read from this file, and prints out the solution (if one exists) using full stops. If the program succeeds it should exit with a status of `0`, or if no route exists it should exit with a status of `1`.

■

```
# # # # # # # # # #
. . # . . . . . . #
# . # . # . # # . #
# . # . # # # # . #
# . # . . . . # . #
# . # . # # # # . #
# . # . . . . # . #
# . # # # # . # . #
# . . . . . . # . .
# # # # # # # # # #
```

6. Lists, Insertion Sort & more Recursion

6.1 A Simple Spelling Checker

Here, an insertion sort involes creating an abstract list data structure, and then reading strings one at a time (possibly from file) and placing them in the **correct** part of the structure. This has a complexty of $O(n^2)$.

For this purpose, a list of valid words (unsorted) is available from the usual place.

Exercise 6.1 Write a program which, based on a list implemented using **arrays**, reads the words in one at a time, inserting them into the **correct** part of the list so that the words are alphabetically sorted. The name of the file should be passed as `argv[1]`, and you can assume the array is of a fixed-size, and large enough to hold all words. How long does it take to build the list ? ■

Exercise 6.2 Now extend Exercise 6.1 so that when the user is prompted for a word, they are told whether this word is present in the list or not. Use a binary search to achieve this. How much faster is this than a linear search ? ■

Exercise 6.3 Now extend Exercise 6.1 so that when the user is prompted for a word, they are told whether this word is present in the list or not. Use an interpolation search to achieve this. How much faster is this than a linear search ? ■

Exercise 6.4 Write a program which, based on a dynamic linked list data structure, reads the words in one at a time, inserting them into the **correct** part of the list so that the words are alphabetically sorted. The name of the file should be passed as `argv[1]`. How long does it take to build the list ? ■

6.2 Prime Factors

www `en.wikipedia.org/wiki/Prime_factor`

It is well known that any positive integer has a single *unique* prime factorization, e.g.:

$210 = 7 \times 5 \times 3 \times 2$ (the numbers $7, 5, 3$ and 2 are all prime).

$117 = 13 \times 3 \times 3$ (the numbers 13 and 3 are all prime).

197 is prime, so has only itself (and 1, which we ignore) as a factor.

Write a program that, for any given positive integer input using `argv[1]`, lists the prime factors, e.g.:

```
[campbell@icy]% ./primefacts 210
7 5 3 2

[campbell@icy]% ./primefacts 117
13 3 3
```

```
72 = 2^3 * 3^2
```

Exercise 6.5 To make the output of the above program briefer, many prefer to show the factors expressed by their power, as in :

$$768 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 3$$

could be better expressed as :

$$768 = 2^8 \times 3$$

Write a program to show the factorisation of a number in this more compact style :

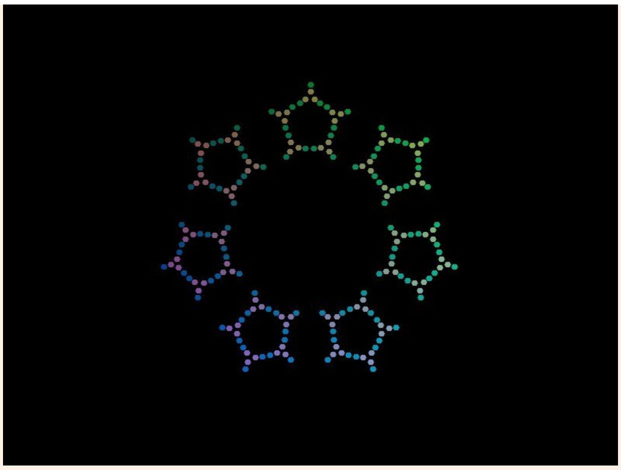
```
 % ./primefactors 27000
27000 = 1 x 2^3 x 3^3 x 5^3
 % ./primefactors 31
31 = 1 x 31
 % ./primefactors 38654705664
38654705664 = 1 x 2^32 x 3^2
```

■

For a beautiful visualisation of prime factors, see:

www www.datapointed.net/visualizations/math/factorization/animated-diagrams

Exercise 6.6 Adapt the program above to output a pattern similar to the animated display above, using SDL, but only for a single number, not an animation.



6.3 Sierpinski Carpet

www en.wikipedia.org/wiki/Sierpinski_carpet

The square is cut into 9 congruent subsquares in a 3-by-3 grid, and the central subsquare is removed. The same procedure is then applied recursively to the remaining 8 subsquares, ad infinitum.

www http://www.evilmadscientist.com/2008/sierpinski-cookies/

Exercise 6.7 Write a program that, in plain text, produces a Sierpinski Carpet.

Exercise 6.8 Write a program that, using ncurses, produces a Sierpinski Carpet.

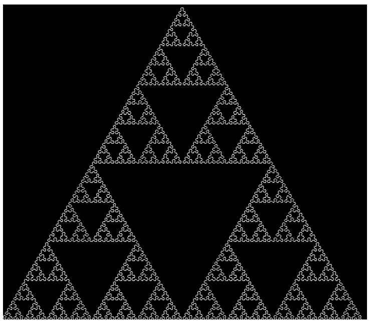
Exercise 6.9 Write a program that, using SDL, produces a Sierpinski Carpet.

6.4 Sierpinski Squares

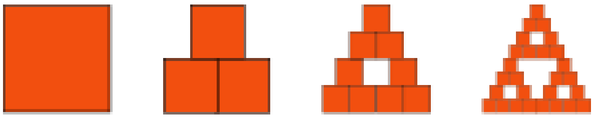
See also :

www en.wikipedia.org/wiki/Sierpinski_triangle

The Sierpinski triangle has the overall shape of an equilateral triangle, recursively subdivded into four smaller triangles :



However, we can approximate it by recursively drawing a square as three smaller squares, as show below :



The recursion should terminate when the squares are too small to draw with any more detail (e.g. one pixel, or one character in size).

Exercise 6.10 Write a program that, in plain text, produces a Sierpinski Triangle. ■

Exercise 6.11 Write a program that, using ncurses, produces a Sierpinski Triangle. ■

Exercise 6.12 Write a program that, using SDL, produces a Sierpinski Triangle. ■

7. Searching Boards

7.1 Conway's Soldiers

The one player game, *Conway's Soldiers* (sometimes known as *Solitaire Army*), is similar to peg solitaire. For this exercise, Conway's board is a 7 (width) × 8 (height) board with tiles on it. The lower half of the board is entirely filled with tiles (pegs), and the upper half is completely empty. A tile can move by jumping another tile, either horizontally or vertically (but never diagonally) onto an empty square. The jumped tile is then removed from the board. A few possible moves are shown below:

```
. . . . . . .    . . . . . . .    . . . . . . .    . . . . . . .
. . . . . . .    . . . . . . .    . . . . . . .    . . . . . . .
. . . . . . .    . . . . . . .    . . . . . . .    . . . . . . .
. . . . . . .    . . X . . . .    . . X X . . .    . . . . X . .
X X X X X X X    X X . X X X X    X X . . X X X    X X . . X X X
X X X X X X X    X X . X X X X    X X . . X X X    X X . . X X X
X X X X X X X    X X X X X X X    X X X X X X X    X X X X X X X
X X X X X X X    X X X X X X X    X X X X X X X    X X X X X X X
```

The user enters the location of an empty square they'd like to get a tile into, and the program demonstrates the moves that enables the tile to reach there (or warns them it's impossible). To do this you will use a list of boards. The initial board is put into this list. Each board in the list is, in turn, read from the list and all possible moves from that board added into the list. The next board is taken, and all its resulting boards are added, and so on.

Each structure in the list will contain (amongst other things) a board and a record of its parent board, i.e. the board that it was created from.

Exercise 7.1 Write a program that:
- Inputs a target location for a tile to reach (x in argv[1], y in argv[2]).
- Demonstrates the correct solution (reverse order is fine) using plain text.

Use the algorithm described above and not anything else.

7.2 The 8-Tile Puzzle

The Chinese 8-Tile Puzzle is a 3×3 board, with 8 numbered tiles in it, and a hole into which neighbouring tiles can move:

1	2	3
4	5	6
7	8	

After the next move the board could look like:

1	2	3
4	5	
7	8	6

or

1	2	3
4	5	6
7		8

The problem generally involves the board starting in a random state, and the user returning the board to the ‘ordered’ ”12345678” state.

In this problem, a solution could be found in many different ways; the solution could be recursive, or you could implement a queue to perform a breadth-first seach, or something more complex allowing a depth-first search to measure ‘how close’ (in some sense) it is to the correct solution.

Exercise 7.2 Read in a board using `argv[1]`, e.g.:

```
$ 8tile "513276 48"
```

To do this you will use a list of boards. The initial board is put into this list. Each board in the list is, in turn, read from the list and all possible moves from that board added into the list. The next board is taken, and all its resulting boards are added, and so on. This is, essentially, a queue.

However, one problem with is that repeated boards may be put into the queue and ‘cycles’ occur. This soon creates an explosively large number of boards (several million). You can solve this by only adding a board into the queue if an identical one does not already exisit in the queue. A linear search is acceptable for this task of identifying duplicates. Each structure in the queue will contain (amongst other things) a board and a record of its parent board, i.e. the board that it was created from.

Be advised that a solution requiring as ‘few’ as 20 moves may take 10’s of minutes to compute. If the search is successful, display the solution to the screen using plain-text.

Use the method described above and only this one. Use a static data structure to acheive this (arrays) and **not** a dynamic method such as linked-lists; a (large) $1D$ array of structures is acceptable. Because this array needs to be so large, it’s best to declare it in `main()` using something like:

```
static boards[NUMBOARDS];
```

■

Exercise 7.3 Repeat Exercise 7.2, but use SDL for the ouput rather than plain-text. ■

Exercise 7.4 Repeat Exercise 7.2, but using a dynamic (linked-list), so that you never have to make any assumptions about the maximum numbers of boards stored. ■

Exercise 7.5 Repeat Exercise 7.4, but using a 5×5 board instead. ■

9. Huffman and Trees

9.1 Depth

The following program builds a binary tree at random:

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

#define STRSIZE 5000

struct node{
        char c;
        struct node *left;
        struct node *right;
};
typedef struct node Node;

Node *MakeNode(char c);
void InsertRandom(Node *t, Node *n);
char *PrintTree(Node *t);

int main(void)
{

   char c;
   Node *head = MakeNode('A');
   Node *n;

   srand(time(NULL));
   for(c = 'B'; c < 'G'; c++){
      n = MakeNode(c);
      InsertRandom(head, n);
   }
   printf("%s\n", PrintTree(head));
   return 0;
}
```

```
Node *MakeNode(char c)
{

    Node *n = (Node *)calloc(1, sizeof(Node));
    assert(n != NULL);
    n->c = c;
    return n;

}

void InsertRandom(Node *t, Node *n)
{

    if((rand()%2) == 0){ /* Left */
        if(t->left == NULL){
            t->left = n;
        }
        else{
            InsertRandom(t->left, n);
        }
    }
    else{ /* Right */
        if(t->right == NULL){
            t->right = n;
        }
        else{
            InsertRandom(t->right, n);
        }
    }

}

char *PrintTree(Node *t)
{

    char *str;

    assert((str = calloc(STRSIZE, sizeof(char))) != NULL);
    if(t == NULL){
        strcpy(str, "*");
        return str;
    }
    sprintf(str, "%c (%s) (%s)", t->c, PrintTree(t->left), PrintTree(t->right));
    return str;

}
```

Each node of the tree contains one of the characters 'A' ... 'F'. At the end, the tree is printed out in the manner described in the course lectures.

Exercise 9.1 Adapt the code so that the maximum depth of the tree is computed using a recursive function. The maximum depth of the tree is the longest path from the root to a leaf. The depth of a tree containing one node is 1. ■

9.2 Two Trees

Adapt the code shown in Exercise 9.1, so that two random trees are generated.

Exercise 9.2 Write a Boolean function that checks whether two trees are identical or not. ■

9.3 Huffman Encoding

Huffman encoding is commonly used for data compression. Based on the frequency of occurence of characters, you build a tree where rare characters appear at the bottom of the tree, and commonly occuring characters are near the top of the tree.

For an example input text file, a Huffman tree might look something like:

```
010 :       00101 (   5 *  125)
‘ ’ :         110 (   3 *  792)
‘"’ :   111001010 (   9 *   12)
‘’’ :    00100000 (   8 *   15)
‘(’ : 01100000100 (  11 *    2)
‘)’ : 01100001101 (  11 *    2)
‘,’ :     1001001 (   7 *   39)
‘-’ :     0010010 (   7 *   31)
‘.’ :     1001100 (   7 *   40)
‘/’ : 00100110000 (  11 *    1)
‘0’ : 11100110010 (  11 *    3)
‘1’ :    00100010 (   8 *   15)
‘3’ : 01100000101 (  11 *    2)
‘4’ : 01100001001 (  11 *    2)
‘5’ : 11100110011 (  11 *    3)
‘6’ : 01100001000 (  11 *    2)
‘7’ : 01100001100 (  11 *    2)
‘8’ :   001001101 (   9 *    8)
‘9’ :    10010000 (   8 *   18)
‘:’ : 01100001011 (  11 *    2)
‘A’ :    00100111 (   8 *   16)
‘B’ :   111001101 (   9 *   13)
‘C’ :    10011011 (   8 *   22)
‘D’ :   111001110 (   9 *   13)
‘E’ :    10011010 (   8 *   19)
‘F’ :   111001000 (   9 *   11)
‘G’ :  0110000000 (  10 *    4)
‘H’ :  1110011111 (  10 *    7)
‘I’ :  1110010011 (  10 *    6)
‘J’ : 11100111101 (  11 *    3)
‘K’ :  1110010111 (  10 *    6)
‘L’ :    00100011 (   8 *   15)
‘M’ : 11100111100 (  11 *    3)
‘N’ : 01100001010 (  11 *    2)
‘O’ : 01100000111 (  11 *    2)
‘P’ :  1110011000 (  10 *    6)
‘R’ :  0110000111 (  10 *    5)
‘S’ :    10010001 (   8 *   19)
‘T’ :  0010011001 (  10 *    4)
‘U’ :  1110010010 (  10 *    5)
‘W’ :  0110000001 (  10 *    4)
‘a’ :        1010 (   4 *  339)
‘b’ :     1111110 (   7 *   60)
```

```
'c' :       100101 (   6 *   77)
'd' :        01101 (   5 *  143)
'e' :          000 (   3 *  473)
'f' :       100111 (   6 *   84)
'g' :       111000 (   6 *   94)
'h' :        11110 (   5 *  223)
'i' :         0100 (   4 *  266)
'j' :  01100000110 (  11 *    2)
'k' :     00100001 (   8 *   15)
'l' :        10110 (   5 *  176)
'm' :       101111 (   6 *   92)
'n' :         0111 (   4 *  288)
'o' :         0101 (   4 *  269)
'p' :       101110 (   6 *   89)
'q' :  00100110001 (  11 *    2)
'r' :        11101 (   5 *  214)
's' :         0011 (   4 *  260)
't' :         1000 (   4 *  305)
'u' :       111110 (   6 *  108)
'v' :      0110001 (   7 *   37)
'w' :      1111111 (   7 *   60)
'x' :   1110010110 (  10 *    6)
'y' :       011001 (   6 *   72)
2916 bytes
```

Each character is shown, along with its Huffman bit-pattern, the length of the bit-pattern and the frequency of occurrence. At the bottom, the total number of bytes required to compress the file is displayed.

Exercise 9.3 Write a program that reads in a file (argv[1]) and, based on the characters it contains, computes the Huffman tree, displaying it as above. ■

9.4 Binary Tree Visualisation

The course notes showed a simple way to print out integer binary trees in this form :

```
20(10(5(*)(*))(17(*)(*)))(30(21(*)(*))(*))
```

You could also imagine doing the reverse operation, that is reading in a tree in the form above and displaying it in a 'friendlier' style :

```
20----30
|     |
10-17 21
|
5
```

The tree has left branches vertically down the page and right branches horizontally right. Another example is :

```
17(2(*)(3(*)(4(*)(*))))(6(8(*)(*))(*))
```

which is displayed as:

```
17----6
|     |
2-3-4 8
```

The above examples show the most ‘compact’ form of displaying the trees, but you can use simplifying assumptions if you wish:

- The integers stored in the tree are always ≥ 0.
- The integers stored in the tree are 5 characters (or less) in length.
- It is just as valid to print the tree in either of these ways :

```
1-6       00001-00006         00001-------------00006
| |         |     |             |                 |
2 7       00002 00008         00002             00007
|           |                   |
3-4-5     00003-00004-00005   00003-00004-00005
```

Exercise 9.4 Write a program that reads in a tree using `argv[1]` and the tree displayed to `stdout` with no other printing if no error has occurred. ■

Exercise 9.5 Write a program that reads in a tree using `argv[1]` and displays the tree using SDL. ■

10. ADTs and Algorithms I

10.1 Indexed Arrays

In the usual places are the files `arr.h`, `arr.c`, `testarr.c` and a makefile. These files enable you to build, and test, the ADT for a 1D Indexed Array. This simple replacement for C arrays is ‘safe’ in the sense that if you write the array out-of-bounds, it will be automatically resized correctly (using `realloc()`). The interface to this ADT is in `arr.h` and its implementation is in `arr.c`. Typing `make testarr`, compiles these files together with the test file `testarr.c`. Executing `./testarr` should result in all tests passing correctly.

```
% make -f 1d_adt.mk run
Basic Array Tests ... Start
Basic Array Tests ... Stop
```

Exercise 10.1 Build `testarr`, and check that you understand the use of the functions, including initialization, reading, writing and freeing. Use the makefile provided to run the code, and do some memory-leak checking etc. ■

10.2 Sets

Sets are an important concept in Computer Science. They enable the storage of elements (members), guaranteeing that no element appears more than once. Operations on sets include initializing them, copying them, inserting an element, returning their size (cardinality), finding if they contain a particular element, removing an element if it exists, and removing one element from a random position (since sets have no particular ordering, this could be the first element). Other set operations include union (combining two sets to include all elements), and intersection (the set containing elements common to both sets).

WWW https://www.mathsisfun.com/sets/sets-introduction.html

WWW https://en.wikipedia.org/wiki/Set_(mathematics)

The definition of a Set ADT is given in `set.h`, and a file to test it is given in `testset.c`.

Exercise 10.2 Write `set.c`, so that:

```
% make -f set_adt.mk run
./testset
Basic Set Tests ... Start
Basic Set Tests ... Stop
```

works correctly. Your Set ADT will build on top of the Indexed Array ADT introduced in Exercise 10.1. Only write `set.c`. Alter no other files, including `arr.c`, `arr.h`, `set.h` or the Makefile. ■

10.3 Towards Polymorphism

Polymorphism is the concept of writing functions (or ADTs), without needing to specify which particular type is being used/stored. To understand the quicksort algorithm, for instance, doesn't really require you to know whether you're using integers, doubles or some other type. C is not very good at dealing with polymorphism - you'd need something like Python, Java or C++ for that. However, it does allow the use of `void*` pointers for us to approximate it.

Exercise 10.3 Extend the array ADT discussed in Exercise 10.1, so that any type can be used - files `varr.h` and `testvarr.c` are available in the usual place - use the Makefile used there, simply swapping `arr` for `varr` at the top. ■

10.4 Double Hashing

Here we use double hashing, a technique for resolving collisions in a hash table.

Exercise 10.4 Use double hashing to create a spelling checker, which reads in a dictionary file from `argv[1]`, and stores the words.

Make sure the program:

- Use double hashing to achieve this.
- Makes no assumptions about the maximum size of the dictionary files. Choose an initial (prime) array size, created via malloc(). If this gets more than 60% full, creates a new array, roughly twice the size (but still prime). Rehash all the words into this new array from the old one. This may need to be done many times as more and more words are added.
- Uses a hash, and double hash, function of your choosing.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up. Assuming the program works correctly, this number is the only output required from the program.

■

10.5 Separate Chaining

Separate chaining deals with collisions by forming (hopefully small) linked lists out from a base array.

Exercise 10.5 Adapt Exercise 10.4 so that:

- A linked-list style approach is used.
- No assumptions about the maximum size of the dictionary file is made.

- The same hash function as before is used.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up, on average. Assuming the program works correctly, this number is the only output required from the program.

11. ADTs and Algorithms II

11.1 MultiValue Maps

Many data types concern a single value (e.g. a hash table), so that a string (say) acts as both the key (by which we search for the data) and also as the object we need to store (the value). An example of this a spelling checker, where one word is stored (and searched for) at a time. However, sometimes there is a need to store a value based on a particular key - for instance an associative array in Python allows you to perform operations such as :

```
population["Bristol"] = 536000
```

where a value (the number 536000) is stored using the key (the string "Bristol"). One decision you need to make when designing such a data type is whether multiple values are allowed for the same key; in the above example this would make no sense - Bristol can only have one population size. But if you wanted to store people as the key, with their salary as the value, you might need to use a MultiValue Map (MVM) since people can have more than one job.

Here we write the abstract type for a MultiValueMap that stores key-value pairs, where both the key and the value are strings.

Exercise 11.1 The definition of an MVM ADT is given in `mvm.h`, and a file to test it is given in `testmvm.c`. Write `mvm.c`, so that:

```
% make -f mvm_adt.mk
./testmvm
Basic MVM Tests ... Start
Basic MVM Tests ... Stop
```

works correctly. Use a simple linked list for this, inserting new items at the head of the list. Make no changes to any of my files.

Submit : `mvm.c`. ■

11.2 Rhymes

In the usual place is a dictionary which, for every word, lists the phonemes (a code for a distinct unit of sound) used to pronounce that word in American English. In this file the word itself, and its phonemes, are separated by a '#'). For instance:

```
BOY#B OY1
```

shows that the word `BOY` has two phonemes : `B` and `OY1`.

A simple attempt at finding rhymes for boy would match every word that has `OY1` as its final phoneme. This gives you:

```
POLLOI MCVOY LAFOY ALROY ILLINOIS CROIX DECOY REDEPLOY CLOY
LAVOY MOYE LOYE STOY PLOY KNOY EMPLOY ELROY JOY COY LACROIX
DEVROY ENJOY LOY COYE FOYE MOY DOI BROY TOY LABOY ROI HOY
ROYE NEU CROY SOY YOY MCCOY CHOY GOY ROY BOLSHOI MALLOY JOYE
DESTROY DELACROIX(1) DEBOY MCROY CHOI UNDEREMPLOY FLOY MCKOY
TOYE AHOY BOY OYE SGROI FOIE(1) TROY DEPLOY SAVOY UNEMPLOY
SCHEU WOY BOYE HOYE FOY OI HOI KROY EMPLOY(1) FLOURNOY OIE
MCCLOY ANNOY OY DEJOY
```

Using two phonemes to do the matching is too many, since the only matches are for words that have exactly the same pronunciation:

```
LABOY DEBOY BOY BOYE
```

which are not really rhymes, but homophones. Therefore, using the **correct** number of phonemes will be key to finding ‘good’ rhyming words.

Here we will use the MutliValue Map written in Exercise 11.1 to create two maps. An MVM `map1` stores the word (as the key) and its final n phonemes as a single string (the value). Now `map2` stores the word (value), keyed by its final n phonemes as a single string. Looking up the phonemes associated with a word can be done using the word as a key [1] (via `map1`), and looking up a word given its phonemes can be achieved using `map2`.

Exercise 11.2 Read in the dictionary, and for each line in turn, store the data in the two maps. Now for a requested word to be ‘rhymed’, search for its phonemes using `map1` and then search for matching words using `map2`. The number of phonemes specified for this rhyming is given via the command line, as are the words to be rhymed:

```
$ ./homophones -n 3 RHYME
RHYME (R AY1 M): PRIME RHYME ANTICRIME(1) CRIME ANTICRIME
GRIME RIME
```

The `-n` flag specifies the number of phonemes to use (you may assume the number associated with it always is always separated from the flag by a space). If no `-n` flag is given then the value 3 is assumed. It only makes sense to use a value of $n \leq$ the number of phonemes in the two words being checked. If you use a value greater than this, the results are undefined.

The list of words to be matched may be found on the command line:

```
$ ./homophones -n 4 CHRISTMAS PROGRAM PASSING
CHRISTMAS (S M AHO S): ISTHMUS CHRISTMAS CHRISTMAS'
PROGRAM (G R AE2 M): CENTIGRAM ENGRAM HISTOGRAM WOLFGRAM
MONOGRAM LOGOGRAM HOLOGRAM MICROGRAM SONOGRAM ANGIOGRAM
TELEGRAM PROGRAMME ELECTROCARDIOGRAM ELECTROPHORETOGRAM
REPROGRAM MILLIGRAM ANAGRAM PEGRAM POLYGRAM DIAGRAM
EPIGRAM PROGRAM MAILGRAM MILGRAM INGHRAM CABLEGRAM
MAMMOGRAM KILOGRAM
PASSING (AE1 S IHO NG): GASSING MASENG SURPASSING KASSING
```

[1] Strictly speaking, we don’t need the $map1$ to be capable of storing multiple values, since every word in the dictionary is unique. We’ll use it here though for simplicity.

```
    HASSING PASSING AMASSING MASSING CLASSING HARASSING
```

Use the Makefile supplied for this task. Make the output as similar to that shown above as possible.

Submit : `homophones.c`. ■

11.3 Faster MVMs

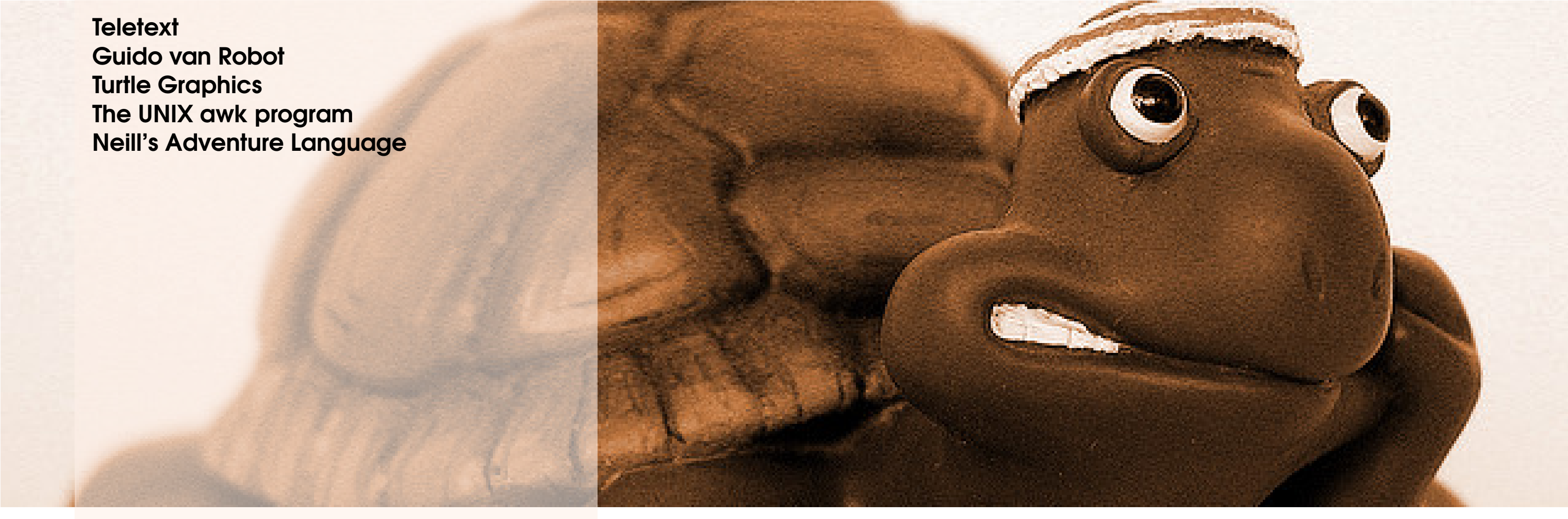
The ADT for MVMs used in Exercise 11.1 is a simple linked list - insertion is fast, but searching is slow.

Exercise 11.3 Write a new version of this MVM ADT called `fmvm.c` that implements exactly the same functionality but has a faster search time. The file `fmvm.h` will change very little from `mvm.h`, with maybe only the structures changing, but not the function prototypes. A similar testing file to that used previously, now called `testfmvm.c` should also be written. Note that any ordering of data when using the `mvm_print` and `mvm_multisearch` functions is acceptable, so these can't be tested in exactly the same manner.

By simply changing the `#include` from `<mvm.h>` to `<fmvm.h>` in your `homephones.c` file from Exercise 11.2, and compiling it against `fmvm.c`, I can test that program works identically.

Make it clear what you have done to speed up your searching (and how) using comments at the top of `fmvm.h`

Submit : `fmvm.c`, `fmvm.h` and `testfmvm.c` ■



12. Parsing Data

12.1 Teletext

In the early 1970s, Phillips Labs began work on transmitting digital information across the television network. The aim was to provide up-to-date news and weather information via a television set. This system was trialled first by the BBC in a system that eventually became known as "Ceefax" and then on other independant British terrestrial stations as "Oracle". A very similar system was implemented on the BBC microcomputer, known as *Mode 7*. An example of

Figure 12.1: An example Ceefax page circa 1983. Taken from `http://teletext.mb21.co.uk/gallery/ceefax/main1.shtml`

	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0	Unused/Reserved	Unused/Reserved		0	@	P	-	p
1	Red Alphanumeric	Red Graphics	!	1	A	Q	a	q
2	Green Alphanumeric	Green Graphics	"	2	B	R	b	r
3	Yellow Alphanumeric	Yellow Graphics	£	3	C	S	c	s
4	Blue Alphanumeric	Blue Graphics	$	4	D	T	d	t
5	Magenta Alphanumeric	Magenta Graphics	%	5	E	U	e	u
6	Cyan Alphanumeric	Cyan Graphics	&	6	F	V	f	v
7	White Alphanumeric	White Graphics	’	7	G	W	g	w
8	Unused/Reserved	Unused/Reserved	(	8	H	X	h	x
9	Unused/Reserved	Contiguous Graphics	)	9	I	Y	i	y
A	Unused/Reserved	Separated Graphics	*	:	J	Z	j	z
B	Unused/Reserved	Unused/Reserved	+	;	K	←	k	¼
C	Single Height	Black Background	,	<	L	½	l	||
D	Double Height	New Background	-	=	M	→	m	¾
E	Unused/Reserved	Hold Graphics	.	>	N	↑	n	÷
F	Unused/Reserved	Release Graphics	/	?	O	#	o	■

Table 12.1: The control codes and characters for alphanumeric mode. Note here (because we’re using white paper) foreground is shown in black and background in white. On a teletext screen we use white on a black background.

such a Ceefax screen is shown in Figure 12.1.

This project, inspired by such teletext systems, will allow a 40×25 (1000) character file to be rendered to the screen, using similar control codes. However, some control codes are not implemented, including those to do with flashing or hidden text, and transparent backgrounds. In particular, our definition of the *double height* control code differs from that of the traditional one.

The Control Codes

This section is based to a large extent to Richard Russell’s description of Mode 7 on the BBC Micro: `http://www.bbcbasic.co.uk/tccgen/manual/tcgen2.html`.

Coloured Text

By using the control codes $129 - 135$ ($0x81 - 0x87$ in hexadecimal) the rest of the line will have text in the selected foreground colour.

To change the background colour, you issue a foreground colour code first, and then the "New Background" character. All the following line will now have the appropriate background colour. You’ll typically then need to choose a new foreground text colour.

Block Graphics

Teletext has a very limited ability to output low-resolution block graphics. These shapes take the place of other characters in the font and are enabled by issuing one of the coloured graphics codes e.g. *red graphics*. At this point the characters available for printing are as displayed in Table 12.2. These new graphics characters are made up of six smaller boxes, known as *sixels*. Each individual sixel has a code, either, $1, 2, 4, 8, 16$ or 64 as shown in Figure 12.2. By adding these values together we can define which of these sixels are ‘lit’ or not. If we wish the three left-hand ones to be lit we’d use the base code (160) plus $1, 4$ and $16 = 181$ ($0xB5$ in hexadecimal). Therefore issuing the coding *green graphics* and then code 181 puts a green vertical bar on the screen.

	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0	Unused/Reserved	Unused/Reserved		🬏	@	P	🬞	🬭
1	Red Alphanumeric	Red Graphics	🬀	🬐	A	Q	🬟	🬮
2	Green Alphanumeric	Green Graphics	🬁	🬑	B	R	🬠	🬯
3	Yellow Alphanumeric	Yellow Graphics	🬂	🬒	C	S	🬡	🬰
4	Blue Alphanumeric	Blue Graphics	🬃	🬓	D	T	🬢	🬱
5	Magenta Alphanumeric	Magenta Graphics	🬄	▌	E	U	🬣	🬲
6	Cyan Alphanumeric	Cyan Graphics	🬅	🬔	F	V	🬤	🬳
7	White Alphanumeric	White Graphics	🬆	🬕	G	W	🬥	🬴
8	Unused/Reserved	Unused/Reserved	🬇	🬖	H	X	🬦	🬵
9	Unused/Reserved	Contiguous Graphics	🬈	🬗	I	Y	🬧	🬶
A	Unused/Reserved	Separated Graphics	🬉	🬘	J	Z	▐	🬷
B	Unused/Reserved	Unused/Reserved	🬊	🬙	K	←	🬨	🬸
C	Single Height	Black Background	🬋	🬚	L	½	🬩	🬹
D	Double Height	New Background	🬌	🬛	M	→	🬪	🬺
E	Unused/Reserved	Hold Graphics	🬍	🬜	N	↑	🬫	🬻
F	Unused/Reserved	Release Graphics	🬎	🬝	O	#	🬬	█

Table 12.2: The control codes and characters for contiguous graphics mode.

	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0	Unused/Reserved	Unused/Reserved			@	P		
1	Red Alphanumeric	Red Graphics			A	Q		
2	Green Alphanumeric	Green Graphics			B	R		
3	Yellow Alphanumeric	Yellow Graphics			C	S		
4	Blue Alphanumeric	Blue Graphics			D	T		
5	Magenta Alphanumeric	Magenta Graphics			E	U		
6	Cyan Alphanumeric	Cyan Graphics			F	V		
7	White Alphanumeric	White Graphics			G	W		
8	Unused/Reserved	Unused/Reserved			H	X		
9	Unused/Reserved	Contiguous Graphics			I	Y		
A	Unused/Reserved	Separated Graphics			J	Z		
B	Unused/Reserved	Unused/Reserved			K	←		
C	Single Height	Black Background			L	½		
D	Double Height	New Background			M	→		
E	Unused/Reserved	Hold Graphics			N	↑		
F	Unused/Reserved	Release Graphics			O	#		

Table 12.3: The control codes and characters for separated graphics mode.

1	2
4	8
16	**64**

Figure 12.2: Values for computing graphics codes, as added to the base code 160 ($0xA0$ in hexadecimal).

Notice in Table 12.2 that some other characters are still available, particularly all capital letters. This allow simple printing of capitals, even when in graphics mode, and is know as *blast-through Text*.

There is another set of block graphics, as shown in Table 12.3. For these, each sixel is separated from others by thin vertical and horizontal lines. This mode is known as *separated graphics* mode.

Held Graphics

Generally all control codes are displayed as spaces, in the current background colour. In the held graphics mode, control code 158 ($0x9E$ in hexadecimal), control codes are instead displayed as a copy of the most recently displayed graphics symbol. This permits a limited range of abrupt display colour changes. The held graphics character is displayed in the same contiguous/separated mode as when it was first displayed. If there has been a change in the text/graphics mode or the normal/double-height mode since the last graphics character was displayed, the held graphics character is cleared and control codes once again display as spaces.

To switch held graphics mode off, use the *release graphics* control code.

Double Height

By using the *double height* control code, characters are displayed as twice their normal size. Since they span two lines, the control codes and characters have to be repeated on the next line too, for them to be correctly displayed. The rule here, is that if a character is to be displayed as double height, the top half of the character is displayed on the first line, and the bottom half on the next line. The bottom half is only displayed as double height if the character vertically above it was also displayed in *double height* mode. The character in question need not be the same one.

Note: here we deviate from other definitions of this control code.

Some General Guidelines

- Characters are considered 7-bit (the 8th bit was typically used for parity checking over the noisy television signal). Therefore any character less than $128(0x80)$ should have 128 added to it. For, example if you read in character 32 (space), it should be 'converted' to character 160.
- Each newline on the Teletext page automatically begins with White text, single height, contiguous graphics, black background, release graphics.
- With the the exception of *hold graphics* (see above), control characters are generally rendered in the same manner as a space would be. If the background is currently red and text colour yellow, say, then the control code would show as an empty red background.

Exercise 12.1 Implement a teletext rendering system. The 1000 character input file should be read in using `argv[1]`.

There are many ambiguities as to how various sequences of codes should be rendered. To help with this, several example files have been posted on the unit web page. If there is still doubt, make a best-guess and state your assumptions in the code.

Submit the testing you have undertaken, including examples and a description of your strategies. This should convince us that you have tested every line of code, and that it works correctly. If there are still issues/bugs state them clearly. Also, point out any bugs that you have successfully found using these approaches. Submit a file named `testing.txt`, along with any other files you feel necessary in the appropriate directory.

No particular strategy is mandated. You may wish to explore a couple and briefly discuss strengths and weaknesses.

Undertake an extension of your choosing. Examples of these include:

- A system that allows you to quickly author teletext pages (perhaps using a recursive-descent parser?)
- Automatic image to teletext conversion.
- Automatic (simple) html to teletext conversion (and/or vice-versa).

Remember, that the assessment is based on the quality of your coding, so choose something to demonstrate an aspect of programming or software engineering that you haven't had a chance to use in the main assignment. Submit a file named `extension.txt` outlining, in brief, your contribution.

Hints

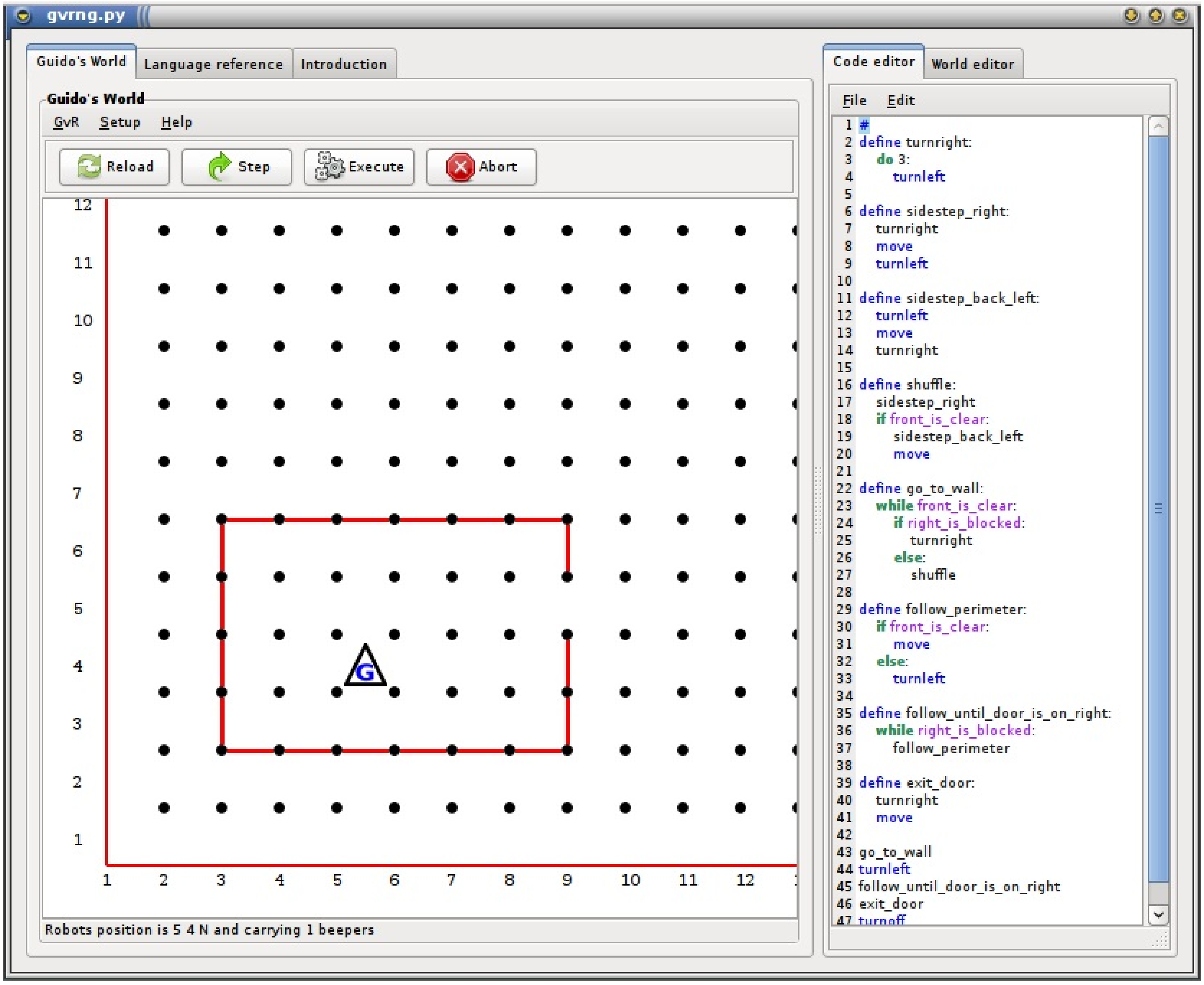
- Don't add graphics too early - the code is easier to test and debug with textual output to begin with.
- I advise you to use SDL for your graphics output. The library provided previously contained two functions to deal with printing characters : `Neill_SDL_ReadFont()` and `Neill_SDL_DrawString()`. The font file `m7fixed.fnt` provides the basic characters required here, but not the sixels. By understanding how the font data is rendered, the double height version of the characters should be relatively simple.
- Don't try to do all aspects of this at once - begin with coloured characters only. Add more advanced functionality later.
- Plan how you are going to store your data early in the design process. Does each character need its own data structure? Does each line? Can this be abstracted?

Please create a directory structure, so that I can easily find the different subsections. Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the source and extension sections, make sure there's a `Makefile`, so that I can easily build the code.

```
        ------Top Directory------
        |            |          |
        |            |          |
        |            |          |
        |            |          |
      source      testing    extension
        |            |          |
       ...          ...   extension.txt
       ...          ...        ...
     Makefile     Makefile     ...
                testing.txt
```

Bundle all of these up as one **single** `.zip` submission - not one for each subsection.

12.2 Guido van Robot



From

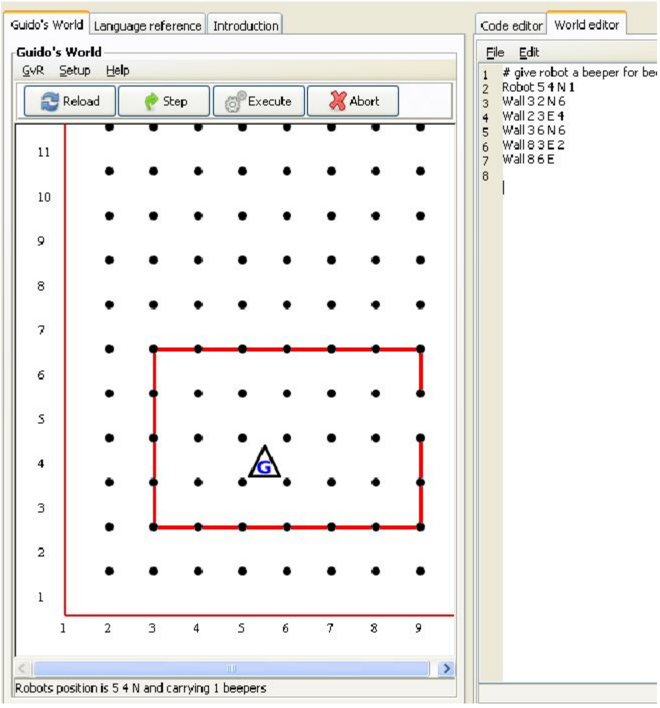
www `http://gvr.sourceforge.net/`

Guido van Robot can face in one of four directions, north, east, south, and west. He turns only 90 degrees at a time, so he can't face northeast, for instance. In Guido's world, streets run east-west, and are numbered starting at 1. There are no zero or negative street numbers. Avenues run north-south, and are also numbered starting at 1, with no zero or negative avenue numbers. At the intersection of a street and avenue is a corner. Guido moves from one corner to to the next in a single movement. Because he can only face in one of four directions, when he moves he changes his location by one avenue, or by one street, but not both!

Simple .wld File

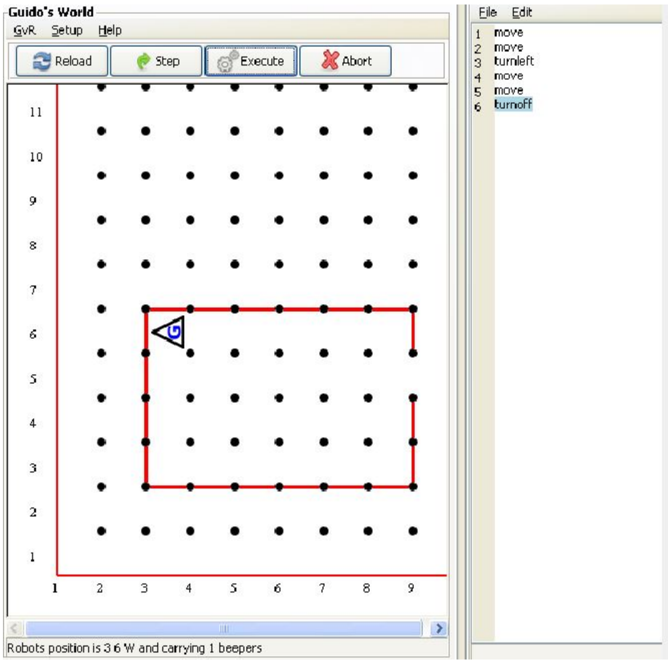
```
Robot 5 4 N 1
Wall 3 2 N 6
Wall 2 3 E 4
Wall 3 6 N 6
```

```
Wall 8 3 E 2
Wall 8 6 E
```



Simple .gvr File

```
move
move
move
move
turnoff
```



Do Loops

```
do 2 :
   putbeeper
   move
turnoff
```

Conditional Loop

```
while front_is_clear :
   putbeeper
   move
turnoff
```

Branching

```
do 13 :
   if front_is_clear :
      putbeeper
      move
   else :
      turnleft
turnoff
```

The Formal Grammar

```
<PROGRAM>   ::= <BLOCK>
<BLOCK>     ::= "turnoff" |
                "move" <BLOCK> |
                "turnleft" <BLOCK> |
                "pickbeeper" <BLOCK> |
                "putbeeper" <BLOCK> |
                <DO> <BLOCK> |
                <WHILE> <BLOCK> |
                <IF> <BLOCK>
<DO>        ::= "do" <num> ":"
                   <BLOCK>
<WHILE>     ::= "while" <TEST> ":"
                   <BLOCK>
<IF>        ::= "if" <TEST> ":"
                   <BLOCK> |
              "if" <TEST> ":"
                   <BLOCK>
              "else" ":"
                   <BLOCK>
<TEST>      ::= <WALL> | <BEEP> | <COMPASS>

<WALL>      ::= "front_is_clear" |
                "front_is_blocked" |
                "left_is_clear" |
                "left_is_blocked" |
                "right_is_clear" |
                "right_is_blocked"
<BEEP>      ::= "next_to_a_beeper" |
                "not_next_to_a_beeper" |
                "any_beepers_in_beeper_bag" |
```

```
                  "no_beepers_in_beeper_bag"
<COMPASS>    ::= "facing_north" |
                  "not_facing_north" |
                  "facing_south" |
                  "not_facing_south" |
                  "facing_east" |
                  "not_facing_east" |
                  "facing_west" |
                  "not_facing_west"
```

This ignores some Guido instructions, e.g. `elseif` and `define`. It also doesn't well explain how to spot the end of a `DO` etc. which is marked by a reduction in indentation. The definition of `.wld` files is so simple a recursive descent parser (and hence grammar) is not required.

Exercise 12.2

- (25%) To implement a recursive descent parser - this says whether or not the given `.gvr` and `.wld` follow the formal grammar or not. The input files are specified via `argv[1]` (`.gvr`) and `argv[2]` (`.wld`) .
- (25%) To implement an interpreter, so that the instructions are executed. Printing the world and robot to screen using simple characters is fine, but many will wish to use SDL.
- (25%) To show a testing strategy on the above - you should give details of white-box and black-box testing done on your code. Describe any test-harnesses used. Give examples of the output of many different turtle programs. Convince me that every line of your C code has been tested.
- (25%) To show an extension to the project in a direction of your choice. It should demonstrate your understanding of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar.

  Submit the program(s) and a Makefile so that I can:
- Compile the parser by typing 'make parse'.
- Compile the interpreter by typing 'make interp'.
- Compile the extension by typing 'make extension'.
- Submit a test strategy report called test.txt. This will include sample outputs, any code written especially for testing etc.
- Submit an extension report called 'extend.txt'. This is quite brief and explains the extension attempted.
- You need to be able to load a world file and a `.gvr` and say if they are valid of not.
- Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

```
<PROGRAM>    ::= <BLOCK>
<BLOCK>      ::= "turnoff" |
                 "move" <BLOCK> |
                 "turnleft" <BLOCK> |
                 "pickbeeper" <BLOCK> |
                 "putbeeper" <BLOCK>
<DO>         ::= "do" <num> ":"
                    <BLOCK>
```

- Some issues, such as what happens if you hit a wall are not clear from the formal grammar. In this case, use your common sense, or do what the real program does.

■

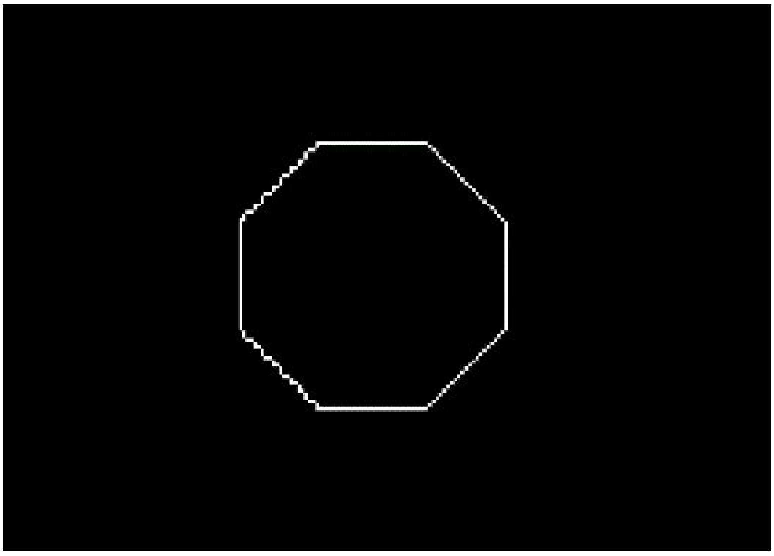
12.3 Turtle Graphics

History

- Many attempts have been made to create programming languages which are intuitive and easy to learn.
- One of the best of these was *LOGO* which allowed children as young as 3 to learn a computer language.
- A subset of this language involved a "turtle" which could be driven around the screen using simple instructions. The turtle, when viewed from above, was represented by a triangle.

An Example

```
{
 FD 30
 LT 45
 FD 30
 LT 45
 FD 30
 LT 45
 FD 30
 LT 45
 FD 30
 LT 45
 FD 30
 LT 45
 FD 30
 LT 45
 FD 30
 LT 45
}
```



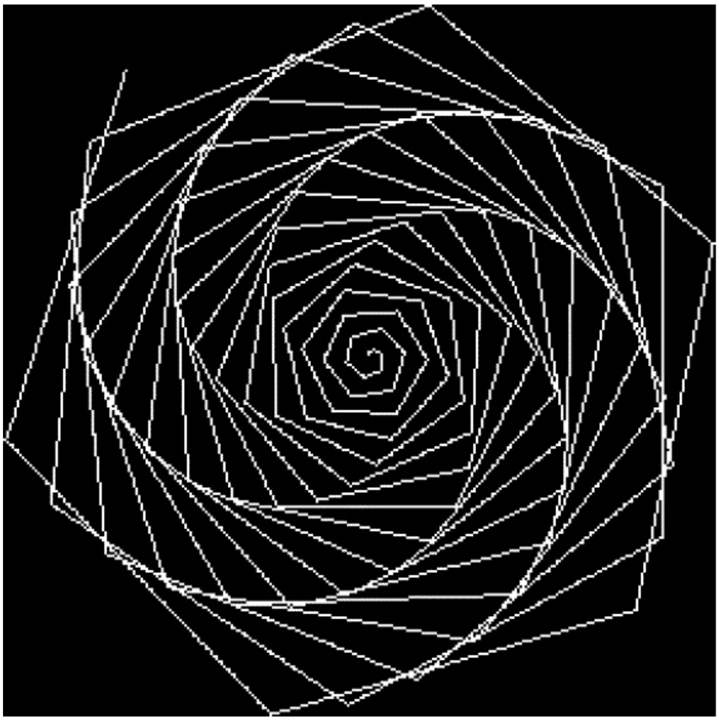
Adding Loops

```
{
   DO A FROM 1 TO 8 {
      FD 30
```

```
        LT 45
    }
}
```

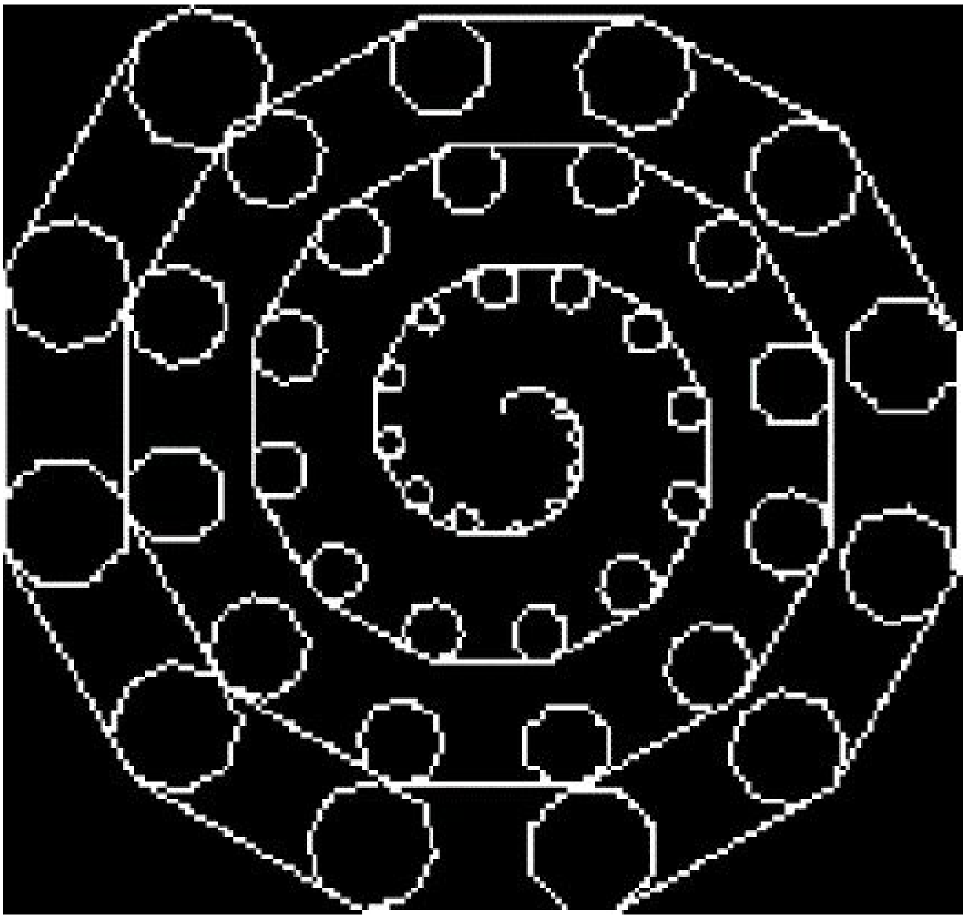
Using Variables

```
{
    DO A FROM 1 TO 100 {
        SET C := A 1.5 ∗ ;
        FD C
        RT 62
    }
}
```



Nested Loops

```
{
    DO A FROM 1 TO 50 {
        FD A
        RT 30
        DO B FROM 1 TO 8 {
            SET C := A 5 / ;
            FD C
            RT 45
        }
    }
}
```



The Formal Grammar

```
<MAIN> ::= "{" <INSTRCTLST>
<INSTRCTLST> ::= <INSTRUCTION><INSTRCTLST> |
                 "}"
<INSTRUCTION> ::= <FD> |
                  <LT> |
                  <RT> |
                  <DO> |
                  <SET>
<FD> ::= "FD" <VARNUM>
<LT> ::= "LT" <VARNUM>
<RT> ::= "RT" <VARNUM>
<DO> ::= "DO" <VAR> "FROM" <VARNUM> "TO"
                  <VARNUM> "{" <INSTRCTLST>
<VAR> ::= [A−Z]
<VARNUM> ::= number | <VAR>
<SET> ::= "SET" <VAR> ":=" <POLISH>
<POLISH> ::= <OP> <POLISH> | <VARNUM> <POLISH> | ";"
<OP> ::= "+" | "−" | "∗" | "/"
```

Exercise 12.3 Implement a recursive descent parser - this will report whether or not a given turtle program follows the formal grammar or not. The input file is specified via `argv[1]` - there is **no** output if the input file is **valid**. Elsewise, a non-zero `exit` is made.

Extend the parser, so it becomes an interpreter. The instructions are now ‘executed’. Do not write a new program for this, simply extend your existing parser. Output is via SDL. You may find the function call `SDL_RenderDrawLine` useful.

Show a testing strategy on the above - you should give details of unit testing, white/black-box testing done on your code. Describe any test-harnesses used. In addition, give examples of the output of many different turtle programs. Convince me that every line of your C code has been tested.

Show an extension to the project in a direction of your choice. It should demonstrate your **understanding** of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar.

Hints

- All four sections above are equally weighted.
- Don't try to write the entire program in one go. Try a cut down version of the grammar first, e.g.:

```
<MAIN>        ::= "{" <INSTRCTLST>
<INSTRCTLST>  ::= <INSTRUCTION><INSTRCTLST> |
                  "}"
<INSTRUCTION> ::= <FD> | <LT> | <RT>
<FD>          ::= "FD" <VARNUM>
<LT>          ::= "LT" <VARNUM>
<RT>          ::= "RT" <VARNUM>
<VARNUM>      ::= number
```

- The language is simply a sequence of words (even the semi-colons), so use `fscanf()`.
- Some issues, such as what happens if you use an undefined variable, or if you use a variable before it is set, are not explained by the formal grammar. Use your own common-sense, and explain what you have done.
- Once your parser works, extend it to become an interpreter. DO NOT aim to parse the program first and then interpret it separately. Interpreting and parsing are inseparably bound together.

Submission

Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the parser, interpreter and extension sections, make sure there's a `Makefile`, so that I can easily build the code using `make parse`, `make interp` and `make extension`.

12.4 The UNIX awk program

Sometimes handling files containing numerical data in C may be somewhat arduous. A 'simple' program to swap the first and second columns of a file is quite long in C.

For this reason, there is a simple language called awk which allows simple manipulation to be done on a line by line basis. For example:

```
{
print $2, $1;
}
```

swaps the first and second fields in a file. The whole program is applied to every line of the input file in turn.

Other Examples of AWK

Printing fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Adding up first column, print sum and average

```
{ s += $1 }
END{print "sum is",s," average is",s/NR}
```

Notice the 'special' variables NR and NF. Other special variables include FS which defines how fields are separated (default space and tab). See 'man awk' for more information.

The CAWK formal grammar

This assignment involves writing a recursive descent parser for a simple, cut-down version of awk called 'cawk'. It is based on the following formal grammar:

```
<PROG>       := "{" <ILST> |
                "{" <ILST> "END" "{" <ILST>
<ILST>       := <INSTR> <ILST> | "}"
<INSTR>      := <LET> | <IF> | <PRINT> | <FOR>
<LET>        := "LET" <USER> "=" <POLISH>
<USER>       : = "$A" | .. | "$Z"
<RD>         := <USER> | "$[" <INDEX> "]" | <SPEC>
<INDEX>      := number | <USER> | <SPEC>
<SPEC>       := "$#" | "$@"
<NUMVAR>     := number | <RD>
<POLISH>     := <OPER><POLISH> | ";"
<OPER>       := <NUMVAR> | <OPERATOR>
<OPERATOR>   := "+" | "-" | "*" | "/"
<IF>         := "IF" "(" <TEST> ")" "{" <ILST>
<TEST>       := <NUMVAR> <COND> <NUMVAR>
<COND>       := "<" | ">" | "==" | "!="
<PRINT>      := "PRINT" <NUMVAR> | "PRINT" string
<FOR>        := "FOR" <USER> "=" <NUMVAR> "TO"
                <NUMVAR> "STEP" <NUMVAR> "{" <ILST>
```

Note, you may assume that the program consists of a list of words separated by white-space.

```
$# is the number of records.
$@ is the number of fields in the current record.
$[ 0 ] is the entire line
$[ i ] is the ith field of the record
```

Examples of CAWK

```
{
PRINT $[ 2 ]
PRINT $[ 1 ]
PRINT "\n"
}
```

Print fields in reverse order:

```
{
   FOR $I = $@ TO 1 STEP -1 {
      PRINT $[ $I ]
   }
   PRINT "\n"
}
```

Exercise 12.4 Write a C program to implement the above formal grammar. Your program should read in a cawk program (argv[1]) and expect the data file to be read from standard input (or from argv[2] if specified).

The marks are split as follows:

- (25%) To implement a recursive descent parser - this says whether or not a given CAWK program follows the formal grammar or not.
- (25%) To implement an interpreter, so that the instructions are executed.
- (25%) To show a testing strategy on the above - you should give details of white-box and black-box testing done on your code. Describe any test-harnesses used. Give examples of the output of many different cawk programs.
- (25%) To show an extension to the project in a direction of your choice. It should demonstrate your understanding of some aspect of programming or S/W engineering. If you extend the formal grammar make sure that you show the new, full grammar.

Submit the program(s) and a Makefile so that I can:

- Compile the parser by typing 'make parse'.
- Compile the interpreter by typing 'make interp'.
- Compile the extension by typing 'make extension'.

In addition:

- Submit a test strategy report called test.txt. This will include sample outputs, any code written especially for testing etc.
- Submit an extension report called 'extend.txt'. This is quite brief and explains the extension attempted.

12.5 Neill's Adventure Language

Some of the very earliest computer games were text-based adventures e.g. Colossal Cave

www https://en.wikipedia.org/wiki/Colossal_Cave_Adventure

Here we write a simple language (NAL) which allows us to write (simplified) versions of such games, focussing on setting variables and printing. The grammar is as follows:

```
<PROGRAM> := "{" <INSTRS>
<INSTRS> := "}" | <INSTRUCT> <INSTRS>
<INSTRUCT> := <FILE> | <ABORT> | <INPUT> | <IFCOND> | <INC> | <SET> |
              <JUMP> | <PRINT> | <RND>

% Execute the instructions in file, then return here e.g. :
% FILE "test1.nal"
<FILE> := "FILE" <STRCON>

% Halt/abort all execution right now !
<ABORT> := "ABORT"
```

```
% Fill a number−variable with a number, or 2 string−variables with string :
% IN2STR ( $C, $ZER ) or INNUM ( %NV )
<INPUT> := "IN2STR" "(" <STRVAR> "," <STRVAR> ")" | "INNUM" "(" <NUMVAR> ")"

% Jump to the nth word in this file (the first word is number zero!)
% Brackets count as one word, "things in quotes" count as one word, e.g. :
% JUMP 5
<JUMP> := "JUMP" <NUMCON>

% Output the value of variable, or constant, to screen with (without a linefeed)
<PRINT> := "PRINT" <VARCON>
<PRINTN> := "PRINTN" <VARCON>

% Set a variable to a random number in the range 0 − 99 e.g. :
% RND ( %N )
% Number should be seeded via the clock to be different on successive executions
<RND> := "RND" "(" <NUMVAR> ")"

% If condition/test is true, execute INSTRS after brace, else skip braces
<IFCOND> := <IFEQUAL> "{" <INSTRS> | <IFGREATER> "{" <INSTRS>
<IFEQUAL> := "IFEQUAL" "(" <VARCON> "," <VARCON> ")"
<IFGREATER> := "IFGREATER" "(" <VARCON> "," <VARCON> ")"

% Add 1 to a number−variable e.g. :
% INC ( %ABC )
<INC> := "INC" "(" <NUMVAR> ")"

% Set a variable. All variables are GLOBAL, and persist across the use of FILE etc.
% $A = "Hello" or %B = 17.6
<SET> := <VAR> "=" <VARCON>

% Some helpful variable/constant rules
% (Here ROT18 is ROT13 for letters and rot5 for digits)
<VARCON> := <VAR> | <CON>
<VAR> := <STRVAR> | <NUMVAR>
<CON> := <STRCON> | <NUMCON>
<STRVAR> := $[A−Z]+
<NUMVAR> := %[A−Z]+
<STRCON> := A plain−text string in double−quotes, e.g. "HELLO.TXT",
            or a ROT18 string in hashes e.g. #URYYB.GKG#
<NUMCON> := A number e.g. 14.301
```

Note that string constants can be entered via the use of double quotes, or using hashes to encode strings using ROT18

www `https://en.wikipedia.org/wiki/ROT13`

in which characters are encoded/decoded according to:

```
Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ
ROT13: NOPQRSTUVWXYZABCDEFGHIJKLM
Plain: abcdefghijklmnopqrstuvwxyz
ROT13: nopqrstuvwxyzabcdefghijklm
Plain: 0123456789
ROT5 : 5678901234
```

the algorithm allows obvious ‘spoilers’ to be hidden from users browsing through `.nal` files, but is simple to apply.

A simple program showing the use of assignment, conditionals and ROT18 is shown:

```
{
    $A = "Neill"
    %E = 12.4
    IFEQUAL ( $A , #Arvyy# ) {
        PRINT #Uryyb Jbeyq!#
    }
}
```

Here string-variables (`$`) and number-variables (`#`) are initialised.

The use of ‘JUMP’ is shown next - this allows execution to move a chosen word in the program. Strings inside quotes count as one word, so the following contains six words:

```
{
    PRINT "Warning : Infinite Loop !"
    JUMP 1
}
```

‘FILE’ allows another file to be executed (as <PROG>) and then execution returns to the original when finished. All variables are shared across files and are global:

```
{
    PRINT "In test4, before"
    FILE "test1.nal"
    PRINT "In test4, after"
}
```

‘RND’ sets a variable to a number in the range $0-99$, while ‘INC’ adds one to a number-variable.

```
{
    %C = 0
    RND ( %A )
    PRINT %A
    INC ( %C )
    IFGREATER ( %C , 9 ) {
        ABORT
    }
    JUMP 4
}
```

‘ABORT’ halts execution instantly.

A simple guessing game is shown below:

```
{

    PRINT "I'm thinking of a number (0−99).\nCan you guess it?"
    RND ( %MINE )
    %CNT = 0

    INC ( %CNT )
    PRINT "Type in your guess"
    INNUM ( %GUESS )
    IFGREATER ( %CNT , 7 ) {
        PRINT #Gbb znal gevrf :−(#
        ABORT
```

```
    }
    IFGREATER ( %GUESS , %MINE ) {
        PRINT "Too Big ! Try again ... "
        JUMP 10
    }
    IFGREATER ( %MINE , %GUESS ) {
        PRINT "Too Small ! Try again ... "
        JUMP 10
    }
    IFEQUAL ( %MINE , %GUESS ) {
        PRINT #Lbh thrffrq pbeerpgyl, lbh jva :-)#
        PRINTN "Number of goes = "
        PRINT %CNT
        ABORT
    }

}
```

'INNUM' gets a number (float) from the user, and the use of 'PRINT' (with a newline after), and 'PRINTN' (without a newline after) is demonstrated.

Exercise 12.5

- **(40%)** Implement a parser. The `.nal` file should be read in using `argv[1]`. If the file is parsed correctly, the only out should be:

```
Parsed OK
```

- **(30%)** Implement an interpreter, building on top of the parser in the manner described in the lectures. Do not write a brand new program - interpretation will be done alongside parsing.
- **(20%)** Submit the testing you have undertaken, including examples and a description of your strategies. This should convince us that you have tested every line of code, and that it works correctly. If there are still issues/bugs state them clearly. Also, point out any bugs that you have successfully found using these approaches. Submit a file named `testing.txt`, along with any other files you feel necessary. Due to the recursive nature of this assignment testing is non-trivial - simply submitting many `.nal` files that 'work' is not sufficient. No particular strategy is mandated. You may wish to explore a couple and briefly discuss strengths and weaknesses.
- **(10%)** Undertake an extension of your choosing. Remember, that the assessment is based on the quality of your coding, so choose something to demonstrate an aspect of programming or software engineering that you haven't had a chance to use in the main assignment. Submit a file named `extension.txt` outlining, in brief, your contribution.

Hints

Don't try to write the entire program in one go. Try a cut down version of the grammar first. Build-up from the `01s` example given in lectures.

- Some issues, such as what happens if you use an undefined variable, or if you use a variable before it is set, are not explained by the formal grammar. Use your own common-sense, and explain what you have done.
- Once your parser works, extend it to become an interpreter. DO NOT aim to parse the program first and then interpret it separately. Interpreting and parsing are inseparably bound together.

Submission

Your testing strategy will be explained in `testing.txt`, and your extension as `extension.txt`. For the parser, interpreter and extension sections , make sure there's one `Makefile`, so that I can easily build the code using `make parse`, `make interp` and `make extension`. I've given an example `makefile` in the usual place, but this is an example only - yours may be different. I wrote only one program `nal.c` and built the two different version by setting a `#define` **via the makefile with** `-DINTERP`. Inside the code itself `#ifdef INTERP` and `#endif` are used. Also make sure that basic testing is available using `make testparse` and `make testinterp`.
Place all the files required for your submission in a single `.zip` file called `nal.zip` - this file will not contain other zipped files.

A. Appendix : Lab Tests

Examination Rules

This is an exam. Please:

- Wait to be seated by a Lab. supervisor.
- Do not talk to anyone else.
- Do not use a Web browser, unless instructed to do so (e.g. to submit files).
- Do not use an email connection.
- Do not use other electronic devices such as laptops, tablets or phones.
- Do not look in other peoples directories.
- Do not use code written in 'groups' with others students.
- **DO** use text books, lecture notes, man pages etc.

If you do not obey the above rules you will be asked to leave.

People arriving late may not be allowed into the Lab.

You have one hour. There is additional time in case a server crashes or some other problem occurs.

Submitting Your Work

As you complete each part, submit it online using the Blackboard submission system. Do not submit files that don't work.

For multi-part questions, if you complete Part 1, submit a file called `part1.c`. If you complete Part 2, submit a file called `part2.c` and so on.

No partial marks are available. The only possible scores for each sub-part are pass (100%) or fail (0%).

Code Style

- Your code bf must compile without warnings using the gcc flags:

  ```
  -pedantic -Wall -Wextra -Wfloat-equal -ansi -O2
  ```

- Do **NOT** use any global variables.

A.1 Anagrams

Part 1 (60%)

An anagram is a rearrangement of a word, using all the letters. Two words are said to be anagrams if they have the same characters, but in a different order. For instance the words 'parsley', 'players' and 'replays' are all anagrams of each other.

Since you need to rearrange the words, two identical words, by definition, are not anagrams.

Using the following template, fill in the function `anagram()`, so that the program runs successfully :

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

int anagram(char s1[], char s2[]);

int main(void)
{
   assert(anagram("elvis", "lives") == 1);
   assert(anagram("dreads", "sadder") == 1);
   assert(anagram("replays", "parsley") == 1);
   assert(anagram("listen", "silent") == 1);
   assert(anagram("orchestra", "carthorse") == 1);

   /* Two identical words are not anagrams */
   assert(anagram("elvis", "elvis") == 0);

   assert(anagram("neill", "neil") == 0);
   assert(anagram("neil", "neill") == 0);
   assert(anagram("horse", "short") == 0);

   return 0;
}

int anagram(char s1[], char s2[])
{


}
```

Obviously, your program will need to run for other, unseen but similar, test cases.

Part 2 (40%)

A deranged anagram has two words with the same characters, but the same character does not appear in the same position.

The words 'elvis' and 'lives' are not a derangement since the 's' is in the same position in both words. However, 'dreads' and 'sadder' are, since no letter appears in the same position between the two words.

Extend the program above so that the following assertions, inside `main()` are correct:

```
    assert(derange("elvis", "lives") == 0);
    assert(derange("dreads", "sadder") == 1);
    assert(derange("replays", "parsley") == 1);
    assert(derange("listen", "silent") == 0);
    assert(derange("orchestra", "carthorse") == 1);
```

A.2 Isograms

Part 1 (60%)

An isogram is a word that has no repeating letters. For instance the words ‘graciously’, ‘disgraceful’ and ‘productively’ are all isograms. However, the word ‘dazzlingly’ is not (it contains the letters ‘z’ and ‘l’ twice).

Using the following template, fill in the function `isogram()`, so that the program runs successfully :

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

int isogram(char *s);

int main(void)
{
    assert(isogram("programming") == 0);
    assert(isogram("housewarmings") == 0);
    assert(isogram("abductions") == 1);
    assert(isogram("housewarming") == 1);
    assert(isogram("hydromagnetics") == 1);
    assert(isogram("uncopyrightable") == 1);
    return 0;
}

int isogram(char *s)
{


}
```

Part 2 (40%)

Using the `isogram()` function written above, write a program which finds the *longest* isogram in a file of words. The name of the file is provided via the use of `argv`. On success, the program simply outputs the longest word and its length, nothing else. For example :

```
$ ./parttwo eowl_shuffle.txt
waveringly (10)
```

The file may contain many isograms of (equal) longest length. In this case, outputting any one of them will do.

A.3 Mutating Boards

Part 1 (60%)

Write a **function** that fills up a square board, randomly, with an integer $0 \ldots 9$. Use a:
`#define N 20`
to define the size of the board. Write another function to print the board. The board may look something like:

```
36753562912709360626
18792023759228973612
93194784503610632061
55476569374525474430
78688431492068926649
50487172722610615949
09177115977673656394
81293908850963856115
98481030444476317596
21785741859753883189
64333860488897764303
09254059469224775481
28936802105110850646
25862847240629908131
10340391969338056640
04626756987282996027
61321599149107587048
04296104222055290283
80409196254499360502
94351743146942264128
```

Write a function to 'mutate' the board. Mutating is done like this:

- Choose two random locations which are **horizontally adjacent** (next to each other left-right).
- Swap these two numbers on the board if the left one is greater than the right one, numerically.
- Choose two random locations which are **vertically adjacent** (next to each other up-down).
- Swap these two numbers on the board if the upper one is greater than the lower one, numerically.
- Repeat the above steps (N*N*N) times.

Now print out the board. It should look something like :

```
00000000000001111224
00000001111111233456
00000111112222244456
00001122222333445666
00112222223333555667
01112223333334556678
01112223334445556779
01122333344445556789
01223334444455666789
01223344445556666789
01223344455666667789
01224444456666777889
01234455566677777889
01234555666677788899
01234555666778888899
```

```
12234566677788888999
12345567777888889999
12445677788888899999
34446678889999999999
46678899999999999999
```

Ensure that if you change the size of your array, by changing your `#define` that the program still operates correctly.

Part 2 (40%)

Adapt the code above, so that the algorithm is:

- Choose two numbers at random locations on the board.
- Check that of these two numbers, the one closest to the centre of the array is numerically less than the number furthest away from the centre. If not, swap them.
- Repeat the above steps (N*N*N*N) times.

Once again randomise the array initially, and ensure that after changing your `#define` the program still works correctly.

When $N = 21$, the array may look something like:

```
999998887777788899999
999987666656666788999
998876554444555678899
998665443333444566889
987654333222233456789
876543322112223345678
865443211111112334568
765432111000011234568
765422100000001234567
764321100000001223467
764321100000001123457
764322100000001223467
765422100000001224567
865432110000011234568
865432211111122334568
876543322212223345678
987654333222233456789
988665444333344567889
998876555444455678899
999887666656666789999
999998887777788899999
```