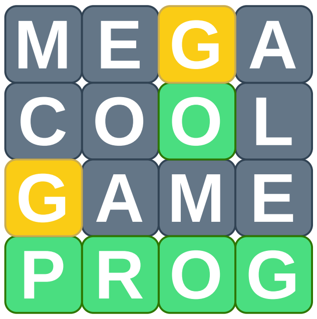
Programming 2 (SS 2022)
Saarland University
Faculty MI
Compiler Design Lab

Project 3

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Marcel Ullrich, B. Sc.

Wordle (15 Points)



Wordle is a game similar to Mastermind. The goal is to guess a word in as few attempts as possible. For each guess, feedback is provided on which characters are correct and at the correct position (marked in green), which characters occur in the final word but at another position (marked in yellow), and which characters do not occur in the final word (marked in gray).
In the example above, the 'G' in the first row is marked yellow as it occurs in `PROG` but at the last position and not the third. The second 'O' in the second row is marked green as it occurs in the same position in `COOL` and in `PROG`. Notice that the first 'O' is marked gray. See below for these special cases.
Further examples and a playable version can be found online[1]

`https://www.nytimes.com/games/wordle/index.html`

Game Principle

The game is determined by the length of the words one plays with and a dictionary containing all possible words of this length. In the usual setting, one plays with a subset of English words where all words are five characters long.
The course of the game is described by

1. Chose a random word from the dictionary
2. Retrieve a valid guess from the player
3. Provide feedback on which characters are `CORRECT` / `WRONGPOS` / `WRONG`
4. Repeat until all characters are correct

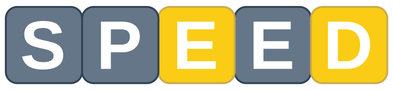
The random word should be chosen uniformly at random from the whole dictionary. Every word should have the same probability of being chosen.
A guess is valid when it is in lower case and is one of the words in the dictionary provided. Therefore, one has to keep track of all possible words and efficiently check whether a word is in the dictionary.

Feedback

After the player has guessed, they are provided feedback on their guess. There, a character at position i is marked green when the character in the secret word at position i coincides with the guess at position i.
Otherwise, the character at position i in guess is marked yellow when the character appears in the secret word at a different position j than in the guess. Additionally, the character in position j in the secret word must not already be used to indicate another position in the guess green or yellow.

[1] If the semantics differ, the semantics described here must be followed.

If neither case applies, the character at position *i* in guess is not marked green nor yellow, it is marked gray.
As an example let us have a look at the word `ABIDE` and the guess `SPEED`.



The first 'E' in column three corresponds to the last letter of `ABIDE` and is thus colored yellow. The second 'E' in column four is the same character as column five in `ABIDE` but the first 'E' in `SPEED` already accounted for the 'E' at position five of `ABIDE`. As there is no second 'E' in `E` the fourth column in `SPEED` is gray.

See below for more details and a formal specification.

Trie

To efficiently implement a check whether the guess is a valid word, a data structure for efficient lookups is needed. Such a data structure is a trie. A trie can be seen as the canonical implementation of the idea to lookup all words with the same first letter, then the word from the results with the same second letter and so on.
One can picture a trie as a tree where each edge is labeled with a character and each node indicates whether a word ends in it.

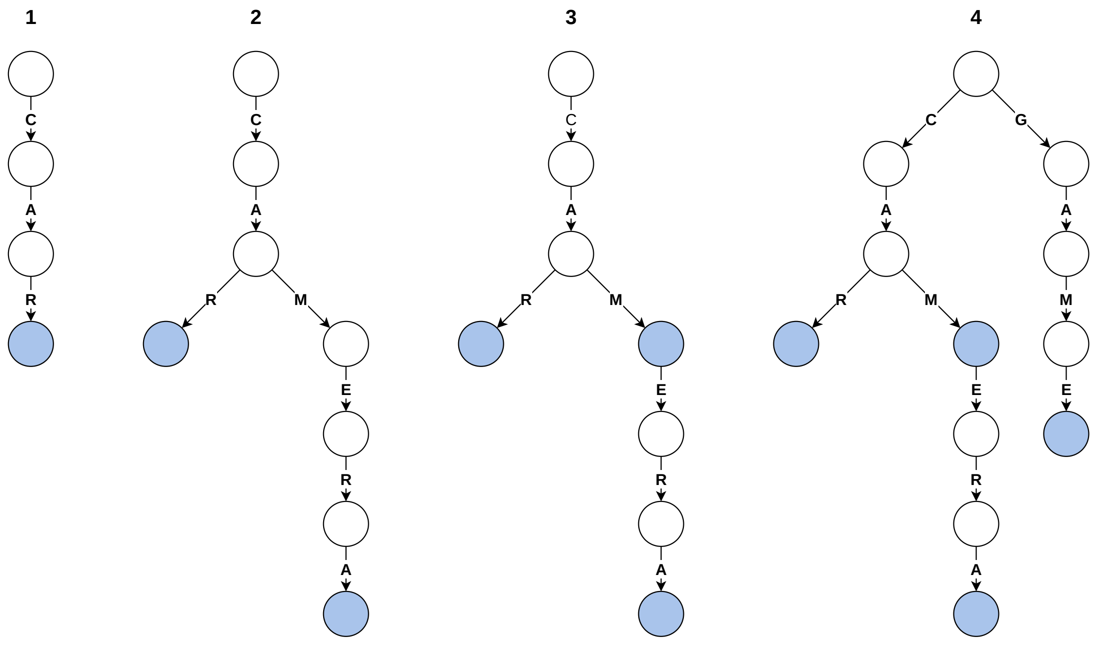


Figure 1: An example construction for a trie. The inserted words are `CAR`, `CAMERA`, `CAM`, and `GAME`.

After construction, one can look up a word by traversing the tree from the root. If the traversal ends in a marked node, the word was found in the dictionary. Otherwise, the word was not in the dictionary.

Tasks

You might want to create multiple header and source files to structure your code. Keep in mind to document your code to understand it better.
Some functions have additional arguments that become important in the quantum wordle implementation.

Trie (5 Points)

Implement a function *create* that creates a new instance of the trie data structure and returns a pointer to it. Initially, the trie should be empty.

Using a function *insert*, a word can be added to the trie. The arguments are a pointer to the trie as well as the string to be inserted.

The *lookup* function also takes the trie and a string. It should return *true* if the string was previously added using *insert* and *false* otherwise.

Lastly, *destroy* should invalidate the trie and free all used memory that was previously allocated in *create*, *insert*, and *lookup*.

You can test your programs to check whether you have a memory leak with

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./bin/wordle_opt [args]
```

Wordle (6 Points)

The *generateDict* function reads a dictionary given a filename and the word length k. It returns a trie containing all words from the dictionary. Each line in the provided file contains exactly one word. The last line might not be terminated using a newline character. You may assume that you only encounter words of length exactly k. Additionally, an initialized pointer is provided as an argument (the second pointer becomes important for the quantum wordle part). A randomly selected word should be written into this pointer. The behavior of the function is undefined if the dictionary is empty. The signature should be as follows: `Trie *generateDict(char *filename, int k, /*@out@*/ char *selected1, /*@out@*/ char *selected2)`
Keep in mind that the word has to be chosen uniformly at random over all words in the dictionary. Use the `drand48()` function. `drand48()` returns a double in the range [0, 1).

guess takes the trie and the length of the word that should be provided by the player. It then asks the user for a guess until a valid one is provided. The prompt should be `"Please input your guess: "` (including the trailing space). A guess is valid if it is in the dictionary. The behavior in the case of non-character inputs or too short inputs is undefined. If an invalid guess is provided, the answer should be `"Invalid word. Try again: "`. Do not repeat the first prompt. All upper case letters should be replaced with their lowercase counterpart. If a valid word is entered, return a pointer to it. The returned word should not include any newlines that may have been entered by the user. The check should always succeed if the dictionary given is a null pointer.

For the feedback, an enumeration is provided with the type name `feedback_result` and the values `CORRECT`, `WRONGPOS`, and `WRONG`.
getFeedback takes the player's guess, the secret word (and a second secret word / nullpointer for qwordle), and the length of the words. It should return a new array containing the feedback if each character from the guess is correct, at a wrong Position, or wrong. Your algorithm should run in time $\mathcal{O}(k)$* for a word of length k.

printFeedback should print `"Result: "` followed by visual feedback for each character and a final newline.
It takes a feedback array and the length of the word. For `CORRECT`, output the unicode sequence 0x0001F7E9 (or 0xD83D 0xDFE9 in UTF-16) also known as 'Large Green Square' 🟩.
For `WRONGPOS`, output the unicode sequence 0x0001F7E8 (or 0xD83D 0xDFE8 in UTF-16) also known as 'Large Yellow Square' 🟨.
For `WRONG`, output the unicode sequence 0x00002B1B (or 0x2B1B in UTF-16) also known as 'Black Large Square' ⬛.

checkWin checks whether the player has won. The function takes a feedback array and the length of the word and returns a boolean.
You have to implement the game logic in *main*. Ask the user for guesses and provide feedback until the player wins.

*The algorithm should have linear runtime in the length of the input. Nested loops, for instance, will most likely be too slow.

Quantum Wordle (3 Points)

We now want to get a bit more advanced and introduce concepts of quantum physics into wordle. There no longer is a single secret word but instead a quantum state of two dual words that represents the secret. Both words have the same length and share no letters. That is, the sets of their characters are disjoint.
The user has to guess either of the two words to win.

If the matched characters are all from the same word, the resulting feedback is identical to normal wordle. The same rules as in normal wordle apply.

If there are matched characters in both words, the feedback uses new quantum states. When a character is in the same position in the guess as in one of the secret words, a quantum correct feedback is returned. When a character is in one of the secret words but not in the same position as in the guess, a quantum wrong position feedback is returned. When a character is in neither of the secret words, a wrong feedback is returned.

Extend all necessary functions to support quantum wordle.
The quantum correct state should be printed using the unicode sequence 0x0001F7E2 (or 0xD83D 0xDFE2 in UTF-16) also known as 'Large Green Circle Emoji' 🟢 .
The quantum wrong position state should be printed using the unicode sequence 0x0001F7E1 (or 0xD83D 0xDFE1 in UTF-16) also known as 'Large Yellow Circle Emoji' 🟡 .

If the *generateDict* function is given a pointer to choose a second word, you have to uniformly at random[2] choose both words such that they fulfill all restrictions. Especially, the two words should not share any characters.
If you want to try quantum wordle online, you can visit the following websites: english version or german version

Code Style (1 Points)

It is important to keep the code readable and easy to understand. This includes writing comments, formatting the code, using functions instead of code duplication, and using indentation. In this project such properties are required and contributed with one point to the overall score.
There are three public tests containing the word `style` that you can run locally to check some of these properties. You should also use the formatting facilities of Visual Studio Code.

Getting Started

To be able to edit the project in Visual Studio Code, you first have to checkout the repository and import the project:

1. Clone the project in any folder:
   `git clone https://prog2scm.cdl.uni-saarland.de/git/project3/$NAME /home/prog2/project3`
   where $NAME has to be replaced with your CMS username.
2. Open the cloned directory in Visual Studio Code and confirm that the contents are trusted.

You can compile the project by executing the command

```
make
```

in the command line in the root of your project folder. You can run the local public tests using

```
make check
```

You can also run single tests using their names. For instance,

```
test/run_tests.py -t "public.<ex>.<name>"
```

runs the test `public.<ex>.<name>`.
Using `-v` as an additional argument, it is also possible to see the called commands. Additionally, you can inspect `src/unit_tests.c` to see the implementation of the public tests.

Apply the patch using `./patch.sh` before starting with the project.

[2]Chose the second word from the set $\{b \mid \text{chars}(b) \cap \text{chars}(a) = \emptyset\}$ for a previously chosen a. If the set is empty, chose another a.

Visual Studio Code Tasks

There are tasks for you to build the project in Visual Studio Code. You can run a task by pressing `F1` and entering `Tasks: Run Task` or simply pressing `F6`. You can run the build task by pressing `Ctrl+Shift+B` or `F7`.
There is a launch configuration to run the program in this project. Select the corresponding launch command from the left toolbar in Visual Studio Code to launch the debugger.

User Interface

You can either play on the command line or using a graphical user interface. Build the project using the command `make`. This will create the executable `bin/wordle_debug` and `bin/wordle_opt`.
Run the game using the command `bin/wordle_debug [k] [dictionary] [quantum]`. To run normal wordle execute `bin/wordle_debug 5 data/5.txt`.

Graphical User Interface

There is a (quantum) wordle interface for you to play the game with. Simply run `./gui.sh` in the command line. This will build the project as a shared library, link it to the GUI, and run it.
Change the arguments in `gui/start.sh` to run with different default parameters.
Note that the GUI might not work even if you have a correct implementation. Conversely, the GUI might work perfectly fine but your project may still have bugs.

Good Luck!

Formal specification

The formal specification of the feedback is split into parts for better readability. In every formula, we quantify over the position i. We use theories in the formulas. But everything could be lowered to first-order logic in a straightforward way.
For every part, we will give the formula and an explanation in natural language.

$$f[i] = \text{CORRECT} \lor f[i] = \text{PRESENT} \lor f[i] = \text{WRONG}$$

The feedback for every character in the guess is either correct, present, or wrong.

$$f[i] = \text{CORRECT} \leftrightarrow g[i] = w[i]$$

A character is correct if and only if g and w coincide.

$$\forall j.\ m[j] = i \rightarrow w[j] = g[i] \land (f[i] = \text{CORRECT} \lor f[i] = \text{PRESENT})$$

Only characters that correspond to the one in guess are marked in the word w. We create a mark for a character exactly when it is correct or present (but not for wrong).

$$\text{someMarked} := \exists j.\ m[j] = i$$

A formula to express that there is a mark for the cell $g[i]$. (The let is implemented using auxiliary variables and equivalence constraints)

$$f[i] = \text{CORRECT} \lor f[i] = \text{PRESENT} \rightarrow \text{someMarked}$$

A correct cell or present cell has to be marked by the specification of marked.

$$\forall j.\ j \neq i \rightarrow m[i] \neq m[j]$$

There can not be two marks for the same cell.

$$m[i] < |g|$$

The marks are negative (for unmarked) or point to a cell in g.

$$g[i] \neq w[i] \land \text{someMarked} \rightarrow f[i] = \text{PRESENT}$$

A marked character in g that is not the same as the corresponding one in w has to be present.

$$\text{noSmallerUnmarked} := \neg\exists i_2.\ i_2 < i \land g[i] = g[i_2] \land \neg\exists l.\ m[l] = i_2$$

There is no previous (smaller position) in g with the same character that is unmarked.

$$f[i] = \text{PRESENT} \rightarrow \text{someMarked} \land \text{noSmallerUnmarked}$$

Recall: For multiple possible cells that can be present, the cells are assigned present from left to right. If there are three 'E' in g which are not correct and there are two 'E' in w, the first two 'E' from g are assigned present. A present cell has to be marked, and no previous cell with the same content can be unmarked.

$$f[i] = \text{WRONG} \rightarrow \forall j.\ w[j] = g[i] \rightarrow m[j] \geq 0 \land m[j] \neq i$$

A wrong char either has no correspondence in w ($\neg(\exists j.\ g[i] \neq w[j])$ which is already included in the second part) or is a failed present. That is, every corresponding character is marked.

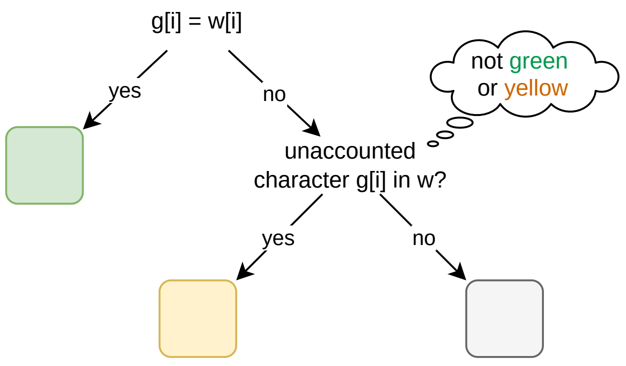
$$\neg(\exists j.\ g[i] \neq w[j]) \rightarrow f[i] = \text{WRONG} \land \neg\text{someMarked}$$

If there is no corresponding character in w, the character in g is assigned wrong.

Whole Formula

$\forall g\ w\ f.\ \exists m.\ \forall i.$

$$
\begin{aligned}
&(f[i] = \text{CORRECT} \lor f[i] = \text{PRESENT} \lor f[i] = \text{WRONG}) \land \\
&(f[i] = \text{CORRECT} \leftrightarrow g[i] = w[i]) \land \\
&(\forall j.\ m[j] = i \rightarrow w[j] = g[i] \land (f[i] = \text{CORRECT} \lor f[i] = \text{PRESENT})) \land \\
&\forall \text{someMarked}.\ (\text{someMarked} \leftrightarrow \exists j.\ m[j] = i) \rightarrow \\
&(f[i] = \text{CORRECT} \lor f[i] = \text{PRESENT} \rightarrow \text{someMarked}) \land \\
&(\forall j.\ j \neq i \rightarrow m[i] \neq m[j]) \land \\
&m[i] < |g| \land \\
&(g[i] \neq w[i] \land \text{someMarked} \rightarrow f[i] = \text{PRESENT}) \land \\
&\forall \text{noSmallerUnmarked}.\ (\text{noSmallerUnmarked} \leftrightarrow \neg\exists i_2.\ i_2 < i \land g[i] = g[i_2] \land \neg\exists l.\ m[l] = i_2) \rightarrow \\
&(f[i] = \text{PRESENT} \rightarrow \text{someMarked} \land \text{noSmallerUnmarked}) \land \\
&(f[i] = \text{WRONG} \rightarrow \forall j.\ w[j] = g[i] \rightarrow m[j] \geq 0 \land m[j] \neq i) \land \\
&(\neg(\exists j.\ g[i] \neq w[j]) \rightarrow f[i] = \text{WRONG} \land \neg\text{someMarked})
\end{aligned}
$$


g[i] = w[i]
yes
no
not green or yellow
unaccounted character g[i] in w?
yes
no