Monash University
Faculty of Information Technology
2nd Semester 2023

FIT2014
...ignment 2
Regular Langu... ...e Languages, Lexical analysis, Parsing,
T... ...d Quantum Computation
...Friday 6 October 2023

In these exercises, you ...

- implement a lex... (Problem 3);
- implement parse... (Problems 1–6);
- program a Turing machine (Problem 7);
- learn about some aspects of quantum circuits and quantum registers, by applying our methods to calculations with them (Problems 2–7);
- practise your skills relating to context-free languages (Problem 8).

Solutions to Problem 7 must be implemented in the simulator Tuatara. We are providing version 2.1 on Moodle under week 8; the file name is `tuatara-monash-2.1.jar`. **You must use exactly this version, not some other version downloaded from the internet.** Do not unpack this file. It must be run directly using the Java runtime.

How to manage this assignment

- You should start working on this assignment now and spread the work over the time until it is due. Do as much as possible *before* week 10.
- Don't be deterred by the length of this document. Much of it is an extended tutorial to get you started with `lex` and `yacc` (pp. 7–11) and documentation for functions, written in C, that are provided for you to use (pp. 12–17); some matrices and sample outputs also take up a fair bit of space. There is an optional three-page introduction to *some* of the basics of quantum computing (pp. 17–19), which is there for those who are interested in knowing more but is not required for this assignment. Although `lex` and `yacc` are new to you, the questions about them only require you to modify some existing input files for them rather than write your own input files from scratch.
- The tasks required for the assignment are on pp. 3–6.
  For Problems 1–5, read the background material on pp. 7–11.
  For Problems 2–6, also read the background material on pp. 12–17.
  For Problem 7, read the background material on p. 21.
  For Problem 8, read the background material on p. 22.

Instructions

Instructions are mostly as for Assignment 1, except that some of the filenames have changed, and now each Problem has its own directory. To begin working on the assignment, download the workbench `asgn2.zip` from Moodle. Create a new Ed Workspace and upload this file, letting Ed automatically extract it. **Edit the `student-id` file to contain your name and student ID.** Refer to Lab 0 for a reminder on how to do these tasks.

Open a terminal and change into the directory `asgn2`. You will find three files already in the directory: `plus-times-power.l`, `plus-times-power.y`, and `quant.h`. You will not modify these files directly; you will make copies of the first two and modify the copies, while `quant.h` must remain unaltered (although it's ok to have copies of it in other directories).

You need to construct new lex files, using `plus-times-power.l` as a starting point, for Problems 1, 3, 4 & 5, and you'll need to construct a new yacc file from `plus-times-power.y` for Problems 4 & 5. Your submission must include:

- the file `student-id`, edited to contain your name and student ID
- a lex file `prob1`[illegible]ained by modifying a copy of `plus-times-power.l`
- a PDF file `prob`[illegible]tain your solution to Problem 2
- a lex file `prob3.`[illegible]obtained by modifying a copy of `plus-times-power.l`
- a lex file `prob4`[illegible]ained by modifying a copy of `prob3.l`
- a yacc file `prob`[illegible]tained by modifying a copy of `plus-times-power.y`
- a lex file `prob5`[illegible]ained by modifying a copy of `prob4.l`
- a yacc file `prob`[illegible]tained by modifying a copy of `prob4.y`
- a *text* file `prob6.txt` which should contain ten lines, being your solution to Problem 6
- a Tuatara Turing machine file `prob7.tm`
- a PDF file `prob8.pdf` which should contain your solution to Problem 8.

The original directory structure and file locations must be preserved. For each problem, the files you are submitting must be in the corresponding subdirectory, i.e. `problem`x for Problem x. Each of these problem subdirectories contains empty files with the required filenames. These must each be replaced by the files you write, as described above. Before submission, **check** that each of these empty files is, indeed, replaced by your own file.

To submit your work, download the Ed workspace as a zip file by clicking on "Download All" in the file manager panel. The "Download All" option preserves the directory structure of the zip file, which is required to aid the marking process. **You must submit this zip file to Moodle by the deadline given above.**

Assignment tasks

First, read about "Lex, Yacc and the PLUS-TIMES-POWER language" on pp. 7–11.

Problem 1. [2

Construct prob1.l, ...1, so that it can be used with plus-times-power.y to build a parser fo... .

Now refer to the document "Quantum circuits, registers and the language QUANT", pages 12–17. In fact, for Problem 2, you can focus just on pages 15–17 and the language QCIRC.

Problem 2. [3 marks]

Write a Context-Free Grammar for the language QCIRC over the ten-symbol alphabet {I, H, X, Z, CNOT, TOF, ∗, ⊗, (,) }. It can be typed or hand-written, but must be in PDF format and saved as a file prob2.pdf.

Now we use some very basic regular expressions (in the lex file, prob3.l) and a CFG (in the yacc file, prob4.y) to construct a lexical analyser (Problem 3) and a parser (Problem 4) for QCIRC.

Problem 3. [3 marks]

Using the file provided for PLUS-TIMES-POWER as a starting point, construct a lex file, prob3.l, and use it to build a lexical analyser for QCIRC.

You'll need to introduce simple regexps for the various tokens, among other things.

Sample output:

```
$ ./a.out
CNOT* ( H (x)I)
Token: CNOT;  Lexeme: CNOT
Token and Lexeme: *
Token and Lexeme: (
Token: H;  Lexeme: H
Token: KRONECKERPROD;  Lexeme: (x)
Token: I;  Lexeme: I
Token and Lexeme: )
Token and Lexeme: <newline>
```

Control-D

Problem 4. [6 marks]

Make a copy of `prob3.l`, call it `prob4.l`, then modify it so that it can be used with `yacc`. Then construct a `yacc` file `prob4.y` from `plus-times-power.y`. Then use these `lex` and `yacc` files to build a pars[illegible]orrectly evaluate any expression in that language.

Note that you c[illegible] any of the quantum expression functions yourself. They have already [illegible]ograms section of the `yacc` file. The *actions* in your `yacc` file will need [illegible] and you can do that by using the function call for `pow(...)` in `plus-t`[illegible]nplate.

The core of you[illegible]rammar rules in the Rules section, in `yacc` format, with associated acti[illegible] in `plus-times-power.y` as a guide. You also need to do some modifications in the Declarations section; see page 10 and further details below.

When entering your grammar into the Rules section of `prob4.y`, it is best to leave the existing rules for the nonterminal `start` unchanged, as this has some extra stuff designed to allow you to enter a series of separate expressions on separate lines. So, take the Start symbol from your grammar in Problem 2 and represent it by the nonterminal `line` instead of by `start`.

The specific modifications you need to do in the Declarations section should be:

- You need a new `%token` declaration for the tokens `I`, `H`, `X`, `Z`, `CNOT`, `TOF`, and `KRONECKERPROD`. These have the same structure as the line for the `NUMBER` token, except that "`num`" is replaced by "`str`" (since each of the above tokens represents a string, being names of matrices, registers or operations, whereas `NUMBER` represents a number).
- You should include each of our two matrix operations, `*` (multiplication) and `KRONECKERPROD` (the Kronecker product, $\otimes$), in a `%left` or `%right` statement. Such a statement specifies when an operation is left- or right-associative, i.e., whether you do multiple consecutive operations left-to-right or right-to-left. Mathematically, for $*$ and $\otimes$, it doesn't matter. So, for these, you can use either `%left` or `%right`. But you should do one of them, because it helps the parser determine an order in which to do the operations and removes some ambiguity. For operations whose `%left` or `%right` statements are on different lines, the operations with higher precedence are those with higher line numbers (i.e., *later* in the file). Ordinary matrix multiplication should have higher precedence than Kronecker product.
- For nonterminal symbols, you can include a `%type` line that declares its type, i.e., the type of value that is returned when an expression generated from this nonterminal is evaluated. E.g.,

  ```
  %type <qmx> start
  ```

  Here, "`qmx`" is the type name we are using for quantum matrices. The various type names can be seen in the `%union` statement a little earlier in the file. But you do not need to know how that works in order to do this task.
- You should still use `start` as your Start symbol. If you use another name instead, you will need to modify the `%start` line too.

Sample output:

```
$ ./a.out
CNOT* ( H (x)I)
4x4 matrix:
   0.707107    0.000000    0.707107    0.000000
   0.000000    0.707107    0.000000    0.707107
   0.000000    0.707107    0.000000   -0.707107
   0.707107    0.000000   -0.707107    0.000000
```

Control-D

Problem 5. [5 marks]

Make a copy of `prob4.l`, call it `prob5.l`. Also, make a copy of `prob4.y`, call it `prob5.y`. Then modify these `lex` and `yacc` files further to build a parser for QUANT that can correctly evaluate any expression in th[illegible]

Again, the core [illegible]e the grammar rules in the Rules section, in `yacc` format with associ[illegible]y. You will also need new tokens `KET0` and `KET1`, which represent `k0` [illegible]hese tokens need appropriate rules in `prob5.l` and declarations in `pro`[illegible]hem in the grammar.

Problem 6. [5 marks]

In the `problem6` directory you will find a file, `prob6.awk`. This is an `awk` program for converting your student ID number to a particular quantum register expression.

Do this conversion by running this `awk` program using `awk -f prob6.awk`, then (when it waits for your input) entering your student ID number. You will see the quantum register expression appear as output.

(a) Copy this quantum register expression (which will be a member of QUANT) and enter it as the first line in the text file `prob6.txt`.

(b) Run your parser on your expression from (a), and report the result of evaluating it by appending the output to the file `prob6.txt`.

The answer to (a) should be the first line of your file `prob6.txt`. Your answer to (b) should occupy the remaining nine lines. The file should therefore have ten lines altogether. You can use wc prob6.txt to verify this.

For example, if your ID is 12345678, then your ten-line file `prob6.txt` should look like this:


```
 (X (x) Z (x) I) * (CNOT (x) Z) * (X (x) Z (x) I) * (k0 (x) k0 (x) k0)
3-qubit register, 8-element vector:
   0.000000
   0.000000
  -1.000000
   0.000000
   0.000000
   0.000000
   0.000000
   0.000000
```

Turing machines

Now refer to the description of **Pauli matrices and the Pauli product word simplification function** on page 21.

Problem 7. [8 [illegible]

Build, in Tuatara, a [illegible] computes the Pauli product word simplification function, and save the T[illegible] prob7.tm.

Context-Free Languages

Now refer to the description of **Walks** on page 22.

Let SAW be the language of strings over the alphabet $\{\mathtt{N}, \mathtt{S}, \mathtt{E}, \mathtt{W}\}$ that represent self-avoiding walks.

Problem 8. [8 marks]

Prove or disprove: The language SAW is context-free.

Your submission can be typed or hand-written, but it must be in PDF format and saved as a file `prob8.pdf`.

Lex, Yacc and the PLUS-TIMES-POWER language

In this part of the Assignment, you will use the lexical analyser generator `lex`, initially by itself, and then with the par[illegible]

Some useful refere[illegible]

- T. Niemann, *Le*[illegible]://epaperpress.com/lexandyacc/
- Doug Brown, Jo[illegible]ason, *lex and yacc (2nd edn.)*, O'Reilly, 2012.
- the `lex` and `yac`[illegible]

We will illustrate [illegible]ams with a language PLUS-TIMES-POWER based on simple arithmetic expressions involving nonnegative integers, using just addition, multiplication and powers. Then you will use `lex` and `yacc` on some languages related to quantum computing.

PLUS-TIMES-POWER

The language PLUS-TIMES-POWER consists of expressions involving addition, multiplication and powers of nonnegative integers, without any parentheses (except for those required by the function Power). Example expressions include:

$$5+8, \quad 8+5, \quad 3+5*2, \quad 13+8*4+\mathrm{Power}(2,\mathrm{Power}(3,2)), \quad \mathrm{Power}(1,3)+\mathrm{Power}(5,3)+\mathrm{Power}(3,3),$$
$$\mathrm{Power}(999,0), \quad 0+99*0+1, \quad 2014, \quad 10*14+74+10*13*73, \quad 2*3*5*7*11*13*17*19.$$

In these expressions, integers are written in unsigned decimal, with no leading zeros or decimal point (so 2014, 86, 10, 7, and 0 are ok, but +2014, −2014, 86.0, A, 007, and 00 are not).

For lexical analysis, we treat every nonnegative integer as a lexeme for the token NUMBER.

Lex

An input file to `lex` is, by convention, given a name ending in `.l`. Such a file has three parts:

- definitions,
- rules,
- C code.

These are separated by double-percent, `%%`. Comments begin with `/*` and end with `*/`. Any comments are ignored when `lex` is run on the file.

You will find an input file, `plus-times-power.l`, among the files for this Assignment. Study its structure now, identifying the three sections and noticing that various pieces of code have been commented out. Those pieces of code are not needed *yet*, but some will be needed later.

We focus mainly on the Rules section, in the middle of the file. It consists of a series of statements of the form

$$\textit{pattern} \qquad \{\ \textit{action}\ \}$$

where the *pattern* is a regular expression and the *action* consists of instructions, written in C, specifying what to do with text that matches the *pattern*.[2] In our file, each *pattern* represents a set of possible lexemes which we wish to identify. These are:

---

[1] actually, Linux includes more modern implementations of these programs called `flex` and `bison`.

[2] This may seem reminiscent of `awk`, but note that: the pattern is not delimited by slashes, `/.../`, as in `awk`; the *action* code is in C, whereas in `awk` the actions are specified in `awk`'s own language, which has similarities with C but is not the same; and the *action* pertains only to the text that matches the pattern, whereas in `awk` the action pertains to the entire line in which the matching text is found.

- a decimal representation of a nonnegative integer, represented as described above;
  - This is taken to be an instance of the token NUMBER (i.e., a lexeme for that token).
- the specific string `Power`, which is taken to be an instance of the token POWER.
- certain specific characters: `+`, `*`, `(`, `)`, and comma;
- the newline char [illegible]
- white space, bei [illegible] es and tabs.

Note that all matchin [illegible] e.

Our *action* is, in n [illegible] essage saying what token and lexeme have been found. For white space, we ta [illegible] haracter that cannot be matched by any pattern yields an error message.

If you run `lex` on [illegible] `wer.l`, then `lex` generates the C program `lex.yy.c`.[3] This is the source code [illegible] r. You compile it using a C compiler such as `cc`.

For this assignment we use `flex`, a more modern variant of `lex`. We generate the lexical analyser as follows.


```
$ flex plus-times-power.l
$ cc lex.yy.c
```

By default, `cc` puts the executable program in a file usually called `a.out`[4] but sometimes called `a.exe`. This can be executed in the usual way, by just entering `./a.out` at the command line. If you prefer to give the executable program another name, such as `plus-times-power-lex`, then you can tell this to the compiler using the `-o` option: `cc lex.yy.c -o plus-times-power-lex`.

When you run the program, it will initially wait for you to input a line of text to analyse. Do so, pressing Return at the end of the line. Then the lexical analyser will print, to standard output, messages showing how it has analysed your input. The printing of these messages is done by the `printf` statements from the file `plus-times-power.l`. Note how it skips over white space, and only reports on the lexemes and tokens.


```
$ ./a.out
13+8    *   4 + Power(2,Power          (3,2       ))
Token: NUMBER;  Lexeme: 13
Token and Lexeme: +
Token: NUMBER;  Lexeme: 8
Token and Lexeme: *
Token: NUMBER;  Lexeme: 4
Token and Lexeme: +
Token: POWER;  Lexeme: Power
Token and Lexeme: (
Token: NUMBER;  Lexeme: 2
Token and Lexeme: ,
Token: POWER;  Lexeme: Power
Token and Lexeme: (
Token: NUMBER;  Lexeme: 3
Token and Lexeme: ,
Token: NUMBER;  Lexeme: 2
Token and Lexeme: )
Token and Lexeme: )
Token and Lexeme: <newline>
```

Try running this program with some input expressions of your own. You can keep entering new expressions on new lines, and enter Control-D to stop when you are finished.

[3] The C program will have this same name, `lex.yy.c`, regardless of the name you gave to the `lex` input file.
[4] `a.out` is short for *a*ssembler *out*put.

Yacc

We now turn to parsing, using yacc.

Consider the following grammar for PLUS-TIMES-POWER.

$$
\begin{aligned}
S &\longrightarrow E \\
& \quad I \\
& \quad \mathbf{POWER}(E, E) \\
& \quad E * E \\
& \quad E + E \\
& \quad \mathbf{NUMBER}
\end{aligned}
$$

In this grammar, the non-terminals are S, E and I. Treat **NUMBER** and **POWER** as just single tokens, and hence single terminal symbols in this grammar.

We now generate a parser for this grammar, which will also evaluate the expressions, with $+, *$ interpreted as the usual integer arithmetic operations and `Power(...,...)` interpreted as raising its first argument to the power of its second argument.

To generate this parser, you need two files, `prob1.l` (for `lex`) and `plus-times-power.y` (for `yacc`):

- Change into your `problem1` subdirectory and do the following steps in that directory.
- Copy `plus-times-power.l` to a new file `prob1.l`, and then modify `prob1.l` as follows:
  - in the **Definitions** section, **un**comment the statement `#include "y.tab.h"`;
  - in the **Rules** section, in each *action*:
    - **un**comment the statements of the form
      - `yylval.str = ...;`
      - `yylval.num = ...;`
      - `return` *TOKENNAME*`;`
      - `return *yytext;`
      - `yyerror...`
    - Comment out the `printf` statements. These may still be handy if debugging is needed, so don't delete them altogether, but the lexical analyser's main role now is to report the tokens and lexemes to the parser, not to the user.
  - in the **C code** section, comment out the function `main()`, which in this case occupies four lines at the end of the file.
- `plus-times-power.y`, the input file for `yacc`, is provided for you. You don't need to modify this *yet*.

An input file for `yacc` is, by convention, given a name ending in `.y`, and has three parts, very loosely analogous to the three parts of a `lex` file but very different in their details and functionality:

- Declarations,
- Rules,
- Programs.

These are separated by double-percent, `%%`. Comments begin with `/*` and end with `*/`.

Peruse the provided file `plus-times-power.y`, identify its main components, and pay particular attention to the following, since you will need to modify some of them later.

- in the Declarations section:
  - lines like

    ```
    int printMatrix(Matrix x);
    Matrix identity();
    :
    Regist[illegible]Reg(Register v, Register w);
    ```

    which are [illegible] (but they are *defined* later, in the Programs section);[5]
  - declaration[illegible]ed:

    ```
    %toke[illegible]
    %toke[illegible]
    ```

  - some specif[illegible]perations are left-associative (which helps determine the order in which operations are applied and can help resolve conflicts and ambiguities):

    ```
    %left '+'
    %left '*'
    ```

  - declarations of the nonterminal symbols to be used (which don't need to start with an upper-case letter):

    ```
    %type <iValue> start
    %type <iValue> line
    %type <iValue> expr
    %type <iValue> int
    ```

  - nomination of which nonterminal is the Start symbol:

    ```
    %start start
    ```

- in the Rules section, a list of grammar rules in Backus-Naur Form (BNF), except that the colon ":" is used instead of →, and there must be a semicolon at the end of each rule. Rules with a common left-hand-side may be written in the usual compact form, by listing their right-hand-sides separated by vertical bars, and one semicolon at the very end. The terminals may be token names, in which case they must be declared in the Declarations section and also used in the `lex` file, or single characters enclosed in forward-quote symbols. Each rule has an *action*, enclosed in braces {...}. A rule for a Start symbol may print output, but most other rules will have an action of the form `$$ = ...` . The special variable `$$` represents the value to be returned for that rule, and in effect specifies how that rule is to be interpreted for evaluating the expression. The variables `$1`, `$2`, ... refer to the values of the first, second, ... symbols in the right-hand side of the rule.

- in the Programs section, various functions, written in C, that your parsers will be able to use. You do not need to modify these functions, and indeed should not try to do so unless you are an experienced C programmer and know exactly what you are doing! Most of these functions are not used yet; some will only be used later, in Problem 4.

After constructing the new `lex` file `prob1.l` as above, the parser can be generated by:

```
$ yacc -d plus-times-power.y
$ flex prob1.l
$ cc lex.yy.c y.tab.c -lm
```

The executable program, which is now a parser for PLUS-TIMES-POWER, is again named `a.out` by default, and will replace any other program of that name that is sitting in the same directory.

[5] These functions for computing with quantum expressions are not needed by `plus-times-power.y`, but you will need them later, when you make a modified version of `plus-times-power.y` to parse quantum expressions.

```
$ ./a.out
13+8   *  4 + Power(2,Power(3,2))
557
13+8*4+Power(2,Power(3,2))
557
Power(1,3)+Power(5
153
1+2+3+4+5+6+7+8+9+
55
10*9*8*7*6*5*4*3*2
3628800
Power(999,0)
1
```
Control-D

Run it with some input expressions of your own. You can keep entering new expressions on new lines, as above, and enter Control-D to stop when you are finished.

Quantum circuits, registers and the language QUANT

Introduction

Roughly speaking, a **quantum computer** is a computer that is able to use certain capabilities based on quantum ph[illegible] usual ("classical") capabilities that computers have.

The idea to use qu[illegible]utation arose in the 1980s with work by the physicists Richard Feynman and [illegible]t increased considerably in the 1990s when Peter Shor gave an efficient qua[illegible]ger factorisation. It had been assumed that integer factorisation was diffi[illegible] algorithm could be used to break the RSA public-key cryptosystem. RSA [illegible] cryptosystem to be published and is still the most widely used, underpin[illegible]rge proportion of modern electronic communications.[6]. Since then, many qu[illegible] appeared, some improving considerably on the best classical algorithms. [illegible]ct in practice, they will have to be implemented on actual quantum computers and applied to large inputs.

Several dozen quantum computers have been built. Many of these are experimental, while some have been used for highly specialised applications. But they are nowhere near powerful enough or robust enough to be useful on a large scale. Practical quantum computing still faces huge technical challenges including adequately protecting the delicate computations from being perturbed by the surrounding environment. In spite of this, there is considerable optimism about their future, and a widespread belief that, in time, their effect will be revolutionary (as well as some scepticism about this). Although they cannot yet be used to break RSA, the threat they pose has been taken very seriously by cryptographers, whose work has led to the development of new quantum-resistant cryptosystems: `https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms`.

For a recent review of the field's current progress and prospects, see [1].

Mathematically, the heart of a quantum computer has three parts:

- a *register* that stores information, and in fact can store multiple data items simultaneously, in a *superposition*, although these data items cannot be accessed in standard classical ways;
- a *quantum circuit expression*, in which certain basic components (called *gates*) are combined in order to produce a transformation that is applied to the register to produce another register;
- *measurement*, in which some part of a register is measured (i.e., read), but the result is determined probabilistically by the entire contents of the register, and the act of measurement reduces the amount of information held in the register.

In this assignment, we just consider registers and quantum circuit expressions, and in fact we only consider restricted types of these. Nonetheless, our registers and expressions are sufficient, in principle, to represent the pre-measurement parts of quantum computations arbitrarily accurately. We define these concepts now.

A **register** is a real 2^n-dimensional vector of unit length. Examples of registers include:

$$n=0: \quad \begin{pmatrix} 1 \end{pmatrix} \qquad n=0: \quad \begin{pmatrix} -1 \end{pmatrix} \qquad n=1: \quad \begin{pmatrix} 0.6 \\ -0.8 \end{pmatrix}$$

$$n=2: \quad \begin{pmatrix} 1/2 \\ -1/2 \\ -1/2 \\ 1/2 \end{pmatrix} \qquad n=3: \quad \begin{pmatrix} 0.6 \\ 0.2 \\ -0.2 \\ -0.4 \\ 0.2 \\ -0.2 \\ 0.4 \\ -0.4 \end{pmatrix}$$

[6] Shor also gave an efficient quantum algorithm for another number-theoretic problem called discrete log. A fast discrete log algorithm would break the Diffie-Hellman key-exchange scheme, another important cryptographic tool.

There are two particular registers with $n = 1$ that we use repeatedly:

traditional name	our name	vector
	`k0`	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
	`k1`	$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$

To define quantum [illegible] need two different ways of combining matrices: ordinary matrix multiplic[illegible]ation called *Kronecker product* which is probably new to you. We discuss these in turn.

Ordinarily, multiplying two matrices A and B is only defined if the number of columns of A equals the number of rows of B. In this assignment, we introduce a new **error matrix** whose sole role is to indicate that an error has been made, at some stage in a calculation, due to trying to multiply incompatible matrices. It has no rows, columns or entries, and we regard it as having $n = -1$. Once an error matrix appears in a calculation, you cannot get rid of it: the result of any subsequent calculation with it will again be the error matrix. With this little "hack", we can allow *any* two matrices to be multiplied together, we just have to keep in mind the possibility of producing the error matrix and having it "swamp" the remainder of our current calculation.

We now define the Kronecker product.

Let $A = (a_{ij})_{r\times c}$ be an $r \times c$ matrix, with r rows and c columns, whose i, j-entry is a_{ij}. Similarly, let $B = (b_{kl})_{s\times d}$ be an $s \times d$ matrix, with s rows and d columns, whose k, l-entry is b_{kl}. Then the **Kronecker product** $A \otimes B$ is the $rs \times cd$ matrix, with rs rows and cd columns, formed by multiplying each entry of A by a copy of B:

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1c}B \\ a_{21}B & a_{22}B & \cdots & a_{2c}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1}B & a_{r2}B & \cdots & a_{rc}B \end{pmatrix}$$

It can be shown that the entry in row $(i-1)s + k$ and column $(j-1)d + l$ of $A \otimes B$ is $a_{ij}b_{kl}$ (where $1 \le i \le r$, $1 \le j \le c$, $1 \le k \le s$, $1 \le l \le d$). For example, if both A and B are 2×2 matrices, with

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix},$$

then

$$A \otimes B = \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}.$$

Some concrete examples:

Let I be the usual 2×2 identity matrix,

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{1}$$

Define H by

$$H = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}. \tag{2}$$

Then

$$I \otimes I \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad H \otimes H \;=\; \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{pmatrix},$$

$$I \otimes H \;=\; \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}, \qquad H \otimes I \;=\; \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{pmatrix}. \quad (3)$$

The Kronecker product can also be applied to our vectors, which are, after all, just matrices with only one column:

$$\mathtt{k0} \otimes \mathtt{k1} \;=\; \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \qquad \mathtt{k1} \otimes \mathtt{k0} \;=\; \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}.$$

We see from these examples that Kronecker product is not commutative in general.

Note that the Kronecker product is *always* defined, regardless of the dimensions of the matrices.

Quantum gates

We use a small set of fundamental matrices, called **quantum gates**, to help build our expressions. The quantum gates we use are:

$$I \;=\; \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad H \;=\; \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}, \quad X \;=\; \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Z \;=\; \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

$$\mathrm{CNOT} \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \qquad \mathrm{TOF} \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

This set of gates is sufficient to build a quantum circuit to approximate any quantum computation arbitrarily closely. In fact, even the two-gate set $\{H, \mathrm{TOF}\}$ is sufficient for that.

H is known as the **Hadamard gate**, and X and Z are the **Pauli-X** and **Pauli-Z gates**. CNOT is the **controlled-not gate** and TOF is the **Toffoli gate**.

Languages for quantum expressions

Quantum circuit expressions are defined inductively as follows:

- Each of `I`, `H`, `X`, `Z`, `CNOT` and `TOF` is a quantum circuit expression.
- If Q is a quantu[illegible]en so is (Q).
- If P and Q are q[illegible]ons, then so is $P * Q$. (This represents ordinary matrix multiplication.)
- If P and Q are [illegible]ssions, then so is $P \otimes Q$. (This represents Kronecker product.)

This inductive definiti[illegible]tarting point for writing a Context-Free Grammar for these expressions.

Quantum register expressions are defined inductively as follows.

- Each of `k0` and `k1` is a quantum register expression.
- If R is a quantum register expression, then so is (R).
- If R and S are quantum register expressions, then so is $R \otimes S$.
- If Q is a quantum <u>circuit</u> expression and R is a quantum register expression, then $Q * R$ is a quantum register expression.

This inductive definition can also be used as the starting point for writing a Context-Free Grammar.

The languages QCIRC, QREG and QUANT

We define three languages to describe the types of quantum expressions we are working with.

- QCIRC is the language, over the ten-symbol alphabet {`I`, `H`, `X`, `Z`, `CNOT`, `TOF`, `*`, $\otimes$, `(`, `)` }, of valid quantum circuit expressions.
- QREG is the language, over the twelve-symbol alphabet {`I`, `H`, `X`, `Z`, `CNOT`, `TOF`, `k0`, `k1`, `*`, $\otimes$, `(`, `)` }, of valid quantum register expressions.
- QUANT is their union: QUANT = QCIRC $\cup$ QREG.

When representing members of these languages in text, we replace $\otimes$ by the three-letter string `(x)`, which is intended to be as close as we can get, with keyboard characters, to a cross in a circle! (Note that it uses *lower-case* `x`.) Always remember that it is our text representation of the Kronecker product symbol $\otimes$; it is *not* an expression x enclosed in parentheses, and it is *not* an argument x being supplied to some function! In this assignment, we *only* use the lower-case letter `x` in this way, enclosed in *one* pair of parentheses in order to represent $\otimes$.

In lexical analysis, we treat `(x)` as the sole lexeme associated with the token `KRONECKERPROD`. We also treat `k0` and `k1` as the sole lexemes associated with tokens `KET0` and `KET1`, respectively. This follows a widespread convention that token names are upper-case.

`I`, `H`, `X`, `Z`, `CNOT`, `TOF` are tokens representing the names of the matrices we use as quantum gates. Each is also the sole lexeme for its token.

The following table gives some members of QUANT. For each, we indicate in the right column whether it belongs to QCIRC or QREG.

quantum expression	string repr'n	evaluates to	comments
I	[illegible]	see (1)	QCIRC: ✓ QREG: ✗ This is a 2×2 identity matrix and is a valid quantum <u>circuit</u> expression. But it's not a vector, so is not a quantum <u>register</u> expression.
$H \otimes I$	[illegible]	see (3)	QCIRC: ✓ QREG: ✗ This is also a valid quantum circuit expression, but is not a vector.
$\texttt{k0} \otimes \texttt{k1}$	`k0 (x) k1`	$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$	QCIRC: ✗ QREG: ✓
$H * \texttt{k0}$	`H*k0`	$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$	QCIRC: ✗ QREG: ✓
$(H * \texttt{k0}) \otimes (H * \texttt{k1})$	`(H*k0) (x) (H*k1)`	$\begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{pmatrix}$	QCIRC: ✗ QREG: ✓
$(H \otimes H) * (\texttt{k0} \otimes \texttt{k1})$	`(H (x) H) * (k0 (x) k1)`	$\begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{pmatrix}$	QCIRC: ✗ QREG: ✓
$CNOT * \texttt{k1}$	`CNOT*k1`	error	QCIRC: ✗ QREG: ✓ The attempt to multiply 4×4 matrix CNOT by 2-element column vector `k1` results in error. But the string still belongs to QREG.
$(I \otimes I) * H * \texttt{k1}$	`(I (x) I) *H*k1`	error	QCIRC: ✗ QREG: ✓ Again, an attempt at an illegal matrix multiplication results in error. But the string still belongs to QREG.

Some examples of *invalid* quantum expressions (i.e., not members of QUANT):

expression	text repr'n	comment
$H \otimes$ `k0`	`H (x) k0`	This Kronecker product gives a 4×2 matrix. This is not a square matrix, so it's not a quantum circuit expression, and is also not a vector.
`k0` $* H$		...duct is the wrong way round. As it is, it's a 2×4 matrix, so is ... quantum circuit expression nor a quantum register expression. ...ther way round, `H*k0`, see previous table.
$H + I$		... a valid operation in QUANT.
`Power`$(H, 2)$	`Pow`...	...s not valid in QUANT; we only use it in PLUS-TIMES-POWER. ...mas are not used in QUANT.

Quantum computation

The previous sections give as much detail as you need to know, at a minimum, to do the assignment. In this section, we give a very brief introduction to *some* of the concepts of quantum computing. It won't be enough to enable you to go away and write quantum algorithms. But, if you do want to learn how to do that, what you have learned from reading this section and doing this assignment will make it a bit easier to get started.

Let's consider *quantum registers* again. A quantum register has 2^n entries, for some nonnegative integer n. We could index them by the numbers $0, 1, \ldots, 2^n - 1$. We could also index them by strings of n bits, with these strings being the n-bit binary representations of those index numbers (with leading zeros allowed, this time). The following table shows a register with $n = 2$ and four entries $\frac{-1}{\sqrt{2}}, 0, \frac{-1}{2}, \frac{1}{2}$. The register is shown as a four-element vector on the right. You can verify that its length is $(|\frac{-1}{\sqrt{2}}|^2 + 0^2 + |\frac{-1}{2}|^2 + |\frac{1}{2}|^2)^{1/2} = \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{4}\right)^{1/2} = 1^{1/2} = 1$, as required. On the left, we give it in tabular form. The register's entries are in the third column (amplitude), with the first and second columns showing its indices as numbers and bit-strings respectively.

outcome		amplitude	probability
0	`00`	$-\frac{1}{\sqrt{2}}$	$\frac{1}{2}$
1	`01`	0	0
2	`10`	$-\frac{1}{2}$	$\frac{1}{4}$
3	`11`	$\frac{1}{2}$	$\frac{1}{4}$

register as vector

$$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

The bit-strings that index the entries of the quantum register are its **outcomes**. The entries themselves are called the **amplitudes** of the outcomes.[7] So, in the above register, the outcome `00` has amplitude $\frac{-1}{\sqrt{2}}$, while outcome `11` has amplitude $\frac{1}{2}$.

A classical register, in a classical computer, stores just *one* bit-string. But quantum registers are very different. All the 2^n outcomes, of n bits each, exist *simultaneously* in some sense, each with its own amplitude. Outcomes of zero amplitude (like `01` in the above example) are impossible, but all outcomes with nonzero amplitude are present to some extent. We say that the register is in a **superposition** of all its possible outcomes.

In general, the amplitudes may be arbitrary complex numbers of length ≤ 1, subject to the constraint that the sum of the squares of their absolute values is 1 (so that, as a vector of 2^n entries, its length is 1).

A classical register has n positions (for some n). Each position holds one <u>bit</u>, either `0` or `1` (but not both!).

[7] We follow Lipton and Regan [5] in moving freely between (i) the index of an entry in a vector representing a register and (ii) a vector that has 1 at that index and 0 elsewhere.

A quantum register also has n positions (for some n), but because of superposition, the information at that position is not just one single bit. Rather, it is a superposition of bits, the superposition being induced by the register's superposition of outcomes. This superposition of bits, in a particular position, is called a **qubit**, and we say that the register has n **qubits**.

Mathematically, classical registers may be regarded as a special case of quantum registers. A classical register cont[illegible]g is just a quantum register in which that bit-string, as an outcome, has a[illegible]r outcomes have amplitude 0. For example, a two-bit classical register conta[illegible]o the quantum register $\texttt{k0} \otimes \texttt{k1}$. Such a register is called a **basis register**.

We mentioned *me*[illegible]n our summary of the heart of a quantum computer. We don't use it in thi[illegible]e only way we get output from a quantum computer, so let's consider it no[illegible]

In classical comput[illegible]ill recover whatever bit-string is stored in it at the time, and nothing else. Given that a quantum register contains 2^n outcomes *at once* (in a superposition), we might hope to get much more out of it! But it's a bit trickier than that. When we read a quantum register, we are sampling from a *probability distribution* over its outcomes. Each outcome has a probability of being chosen, with that probability being the square of the absolute value of its amplitude. These probabilities do indeed sum to 1, because the length of the register (as a vector) is 1.

The probabilities for each of the outcomes in our example register above are given in the fourth column of the table. If we read this register, we have a probability of $\frac{1}{2}$ of getting `00`. Outcomes `10` and `11` each have a probability of $\frac{1}{4}$, while outcome `01` is impossible.

For reasons related to the underlying physics, the act of reading a register is called **measurement**.

We have so far envisaged measuring (i.e., reading) an entire quantum register. But it is also possible to measure individual qubits, i.e., to read what is at a given position. Suppose we wanted to measure the second qubit in the two-qubit quantum register given above. The probability of it being `0` is the sum of the squares of the absolute values of the amplitudes of those outcomes with second bit 0:

$$\begin{aligned}
\Pr(\text{measurement of 2nd bit gives } \texttt{0}) &= |\text{amplitude of } \texttt{00}|^2 + |\text{amplitude of } \texttt{10}|^2 \\
&= |\tfrac{-1}{\sqrt{2}}|^2 + |\tfrac{-1}{2}|^2 \\
&= \tfrac{3}{4}.
\end{aligned}$$

Similarly,

$$\begin{aligned}
\Pr(\text{measurement of 2nd bit gives } \texttt{1}) &= |\text{amplitude of } \texttt{01}|^2 + |\text{amplitude of } \texttt{11}|^2 \\
&= |0|^2 + |\tfrac{1}{2}|^2 \\
&= \tfrac{1}{4}.
\end{aligned}$$

A quantum computation proceeds by

- starting with a register with a single outcome having amplitude 1 and all other outcomes having amplitude 0 (such a register can be formed from a Kronecker product of copies of `k0` and `k1`);
- applying a quantum circuit — which we model as a quantum circuit expression — to the register, thereby producing a new register;
- measuring one or more qubits of this new register. This becomes the output of the computation.

The initial register is *not* the means of providing input to the quantum computer; it is more analogous to the Start State of an automaton. Input is provided to the quantum computer by using the input in the construction of the quantum circuit. So, that circuit is determined in some computable way by the input. Specification of how the circuit is computed from the input is part of the specification of a quantum algorithm.

Let's look at a couple of very simple quantum computations.

Suppose we have $n = 1$ (one qubit) and start with $\texttt{k0}$. Let's use a quantum circuit consisting solely of H. Then the

$$H*\texttt{k0} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}.$$

Then, if we measure t

$$\text{outcome} = \begin{cases} 0, & \text{with probability } \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}; \\ 1, & \text{with probability } \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2}. \end{cases}$$

So this is like tossing a fair coin.

Now suppose we use a one-qubit circuit consisting of two consecutive applications of H. So the quantum circuit expression is now $H * H$. Starting again with $\texttt{k0}$, we obtain

$$H*H*\texttt{k0} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

So, this time, we just end up with $\texttt{k0}$ again, which has only one possible outcome. Mathematically, this is no surprise because the matrix H is its own inverse: $H * H = I$. Let's consider what happens in more physical terms:

1. one application of H takes us from $\texttt{k0}$, which has only one possible outcome, to a superposition of two outcomes (so H seems to "spread out" the available amplitude across more outcomes);

2. then another application of H — which we might intuitively expect to "spread things out" even further — actually creates "cancellation" so that only one outcome is left standing. This phenomenon is known as **interference**.

Now let's graduate to two qubits and use the quantum circuit $\mathrm{CNOT}*(H\otimes I)$ with initial register $\texttt{k0} \otimes \texttt{k0}$. The final register is

$$\mathrm{CNOT}*(H\otimes I)*(\texttt{k0}\otimes\texttt{k0}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix}.$$

If we measure the final register, it will give outcome $\texttt{00}$ with probability $\frac{1}{2}$ and outcome $\texttt{11}$ with probability $\frac{1}{2}$. Note that, in both these possible outcomes, the left and right bits are identical. So, in the register, the left and right qubits are not independent; in fact, they are **entangled**.

References

[1] Michael Brooks, Quantum computers: what are they good for?, *Nature* **617** (S1-S3) (24 May 2023), `https://doi.org/10.1038/d41586-023-01692-9`. Part of *Nature Spotlight: Quantum Computing*.

[2] David Deutsch, Quantum theory, the Church-Turing thesis, and the universal quantum computer, *Proceedings of the Royal Society of London, Series A* **400** no. 1818 (1985) 97–117.

[3] Richard P. Feynman, Simulating physics with computers, *International Journal of Theoretical Physics* **21** (1982) 467–488.

[4] Richard P. Feynm[illegible]al computers, *Optics News* **11** (February 1985) 11–20.

[5] Richard J. Lipton [illegible]n, *An Introduction to Quantum Algorithms via Linear Algebra (2nd edn.)*[illegible]e, Ma., USA, 2021.

[6] Michael A. Nielse[illegible], *Quantum Computation and Quantum Information*, Cambridge Univ. [illegible]

Pauli matrices and the Pauli product word simplification function

The **Pauli matrices** are the following 2×2 matrices,

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

where $i = \sqrt{-1}$. We [illegible] before, on p. 14. These matrices are named after the physicist Wolfgang Pa[illegible]re used in quantum mechanics. They are commonly used as gates in quant[illegible]

These matrices sat[illegible]ions.

$$\begin{aligned} X^2 = Y^2 = Z^2 &= I, & (4)\\ XYZ &= iI, & (5)\\ XY &= -YX, & (6)\\ XZ &= -ZX, & (7)\\ YZ &= -ZY, & (8) \end{aligned}$$

where, as usual, I is the 2×2 identity matrix. Equation (4) says that each of X, Y, Z is an **involution** (i.e., self-inverse). Equations (6)–(8) say that each pair is **anticommutative**.

Equations (4)–(8) can be used to simplify any product of Pauli matrices. In fact, using just involution, (4), and anticommutativity, (6)–(8), we can reduce any product of Pauli matrices to one of the following sixteen expressions (eight specific products, each of which may or may not be negated):

$$\pm I, \qquad \pm X, \qquad \pm Y, \qquad \pm Z, \qquad \pm XY, \qquad \pm XZ, \qquad \pm YZ, \qquad \pm XYZ.$$

We call these sixteen expressions **standard forms**. For example, the product $ZYYXZZXX$ may be reduced to a standard form as follows.

$$\begin{aligned} ZYYXZZXX &= ZYYXZZ & \text{(using involution for } X\text{, (4))}\\ &= ZYYX & \text{(using involution for } Z\text{, (4))}\\ &= ZX & \text{(using involution for } Y\text{, (4))}\\ &= -XZ. & \text{(using anticommutativity, (7))} \end{aligned}$$

There is usually more than one way to simplify an expression.

We treat the problem of reduction to standard form as a problem of computation on strings.

A **Pauli product word** is a string over the five-symbol alphabet $\{X, Y, Z, +, -\}$ in which

- the first symbol is $+$ or $-$,
- the rest of the string is a nonempty string over alphabet $\{X, Y, Z\}$.

Such a string represents a product of Pauli matrices, with an initial coefficient $+1$ or -1 represented by the sign at the start.

To represent Pauli product words on the tape of a Tuatara Turing machine, we use P for $+$ and M for $-$. So our alphabet becomes {X,Y,Z,P,M}.

The **Pauli product word simplification function** takes, as input, a Pauli product word and produces, as its output, the standard form of that word, obtained by some sequence of applications of (4), (6), (7), (8) in some order (not necessarily in the order listed, and allowing multiple applications of the same rule).

Notes:

- Any input string that is not a Pauli product word must be rejected, by crashing.
- The letter I belongs to the tape alphabet, and is sometimes needed in the output string, but does not belong to the input alphabet.

Walks

Imagine you are standing at the origin on an infinite x, y-plane. A **walk** is a sequence of steps of one unit each in which each step can be in any one of the four directions parallel to the coordinate axes. A walk is represented as a string over the alphabet {N, S, E, W}, in which the letters represent the possible steps according to the following table.

letter	stands for:	represents a step of:
N	North	one unit in the positive y direction
[illegible]	[illegible]	one unit in the negative y direction
[illegible]	[illegible]	one unit in the positive x direction
[illegible]	[illegible]	one unit in the negative x direction

In general, walks are [illegible] of their previous journey. So, for example, they may cross themselves or d[illegible]s.

The **length** of a w[illegible]eps in it, or equivalently, the length of the string used to represent it. The w[illegible]n contains the origin but nothing else, is called **trivial**.

A walk is **closed** i[illegible], i.e., at the origin. Otherwise, it is **open**.

A walk is **self-avoiding** if it never meets itself. So no point is visited more than once by the walk. (This includes the start and end points. A non-trivial self-avoiding walk must be open. The converse does not hold in general.)

Some examples of walks, relating the strings that represent them to the routes they take, are:

string	journey	lines used in diagram	self-avoiding?
ESENWNNEWNWW	$(0,0) \to (1,0) \to (1,-1) \to (2,-1) \to (2,0) \to (1,0) \to (1,1) \to (1,2) \to (2,2) \to (1,2) \to (1,3) \to (0,3) \to (-1,3)$	red, straight	No
NNWWWSSEEE	$(0,0) \to (0,1) \to (0,2) \to (-1,2) \to (-2,2) \to (-3,2) \to (-3,1) \to (-3,0) \to (-2,0) \to (-1,0) \to (0,0)$	green, zigzag	No
SWWSWSEEN	$(0,0) \to (0,-1) \to (-1,-1) \to (-2,-1) \to (-2,-2) \to (-3,-2) \to (-3,-3) \to (-2,-3) \to (-1,-3) \to (-1,-2)$	blue, wavy	Yes
ε	$(0,0)$	yellow dot	Yes

These walks are illustrated in the diagram on the right.