Monash University
Faculty of Information Technology
2nd Semester 2020

FIT2014
Assignment 2
Regular Languages, Context-Free Languages, Lexical analysis, Parsing, Turing machines and Quaternions
DUE: 11:55pm, Friday 23 October 2020

In these exercises, you will

- implement a lexical analyser using `lex` (Problems 2, 4);
- implement parsers using `lex` and `yacc` (Problems 1, 3–6);
- program a Turing machine (Problem 7);
- learn about quaternions, by applying our methods to calculations with them (Problems 3–7);
- practise your skills relating to context-free languages (Problem 8).

Solutions to Problem 7 must be implemented in the simulator **tuataraMonash** (version 2.1 of Tuatara), which is available on Moodle (week 8).

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. There will not be time during your prac class to do the assignment from scratch; there will only be time to get some help and clarification.
- 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. 2–5) and documentation for functions, written in C, that are provided for you to use (pp. 5–7); some sample outputs also take up a fair bit of space. 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.

Instructions

Instructions are as for Assignment 1, except that some of the filenames have changed. The file to download is now `asgn2.tar.gz`, and unpacking it will create the directory `asgn2` within your `FIT2014` directory. You will find three files already in the directory: `plus-times-power.l`, `plus-times-power.y`, and `quat.h`. You will not modify these files directly; you will make copies of the first two and modify the copies, while `quat.h` must remain unaltered in the directory where you do this work.

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

- a `lex` file `prob1.l` which should be obtained by modifying a copy of `plus-times-power.l`
- a *text* file `prob2.txt` which should contain a single line with a regular expression in `lex` syntax
- a PDF file `prob3.pdf` which should contain your solution to Problem 3
- a `lex` file `prob4.l` which should also be obtained by modifying a copy of `plus-times-power.l`
- a `lex` file `prob5.l` which should be obtained by modifying a copy of `prob4.l`

- a `yacc` file `prob5.y` which should be obtained by modifying a copy of `plus-times-power.y`
- a *text* file `prob6.txt` which should contain two 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.

Each of the problem directories under the `asgn2` directory 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:

1. edit Makefile as described in Lab 0,
2. enter the command '`make`' from within the `asgn2` directory,
3. submit the resulting `.tar.gz` file to Moodle.

As last time, make sure that you have tested the submission mechanism and that you understand the effect of `make` on your directory tree.

INTRODUCTION: Lex, Yacc and the PLUS-TIMES-POWER language

In this part of the Assignment, you will use the lexical analyser generator `lex` or its variant `flex`, initially by itself, and then with the parser generator `yacc`.

Some useful references on Lex and Yacc:

- T. Niemann, *Lex & Yacc Tutorial*, `http://epaperpress.com/lexandyacc/`
- Doug Brown, John Levine, and Tony Mason, *lex and yacc (2nd edn.)*, O'Reilly, 2012.
- the `lex` and `yacc` manpages

We will illustrate the use of these programs with a language PLUS-TIMES-POWER based on simple arithmetic expressions involving nonnegative integers, using just multiplication, division and powers. Then you will use `lex` and `yacc` on a language QUAT of expressions based on quaternions, which we describe later.

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*.[1] In our file, each *pattern* represents a set of possible lexemes which we wish to identify. These are:

- 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: `+`, `*`, `(`, `)`;
- white space, being any sequence of spaces and tabs;
- the newline character.

Note that all matching in `lex` is case-sensitive.

Our *action* is, in most cases, to print a message saying what token and lexeme have been found. For white space, we take no action at all. A character that cannot be matched by any pattern yields an error message.

If you run `lex` on the file `plus-times-power.l`, then `lex` generates the C program `lex.yy.c`.[2] This is the source code for the lexical analyser. You compile it using a C compiler such as `cc`.

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

By default, `cc` puts the executable program in a file usually called `a.out`[3] 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
```

[1]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.

[2]The C program will have this same name, `lex.yy.c`, regardless of the name you gave to the `lex` input file.

[3]`a.out` is short for *a*ssembler *out*put.

```
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.

Yacc

We now turn to parsing, using `yacc`.

Consider the following grammar for PLUS-TIMES-POWER.

$$
\begin{array}{rcl}
S & \longrightarrow & E \\
S & \longrightarrow & \varepsilon \\
E & \longrightarrow & I \\
E & \longrightarrow & \mathbf{POWER}(E,E) \\
E & \longrightarrow & E * E \\
E & \longrightarrow & E + E \\
I & \longrightarrow & \mathbf{NUMBER}
\end{array}
$$

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`):

- 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 printQuaternion(Quaternion);
    Quaternion newQuaternion(double, double, double, double);
    ⋮
    Quaternion rotation(double, Quaternion);
    ```

    which are *declarations* of functions (but they are *defined* later, in the Programs section);[4]
  - declarations of the tokens to be used:

    ```
    %token <num> NUMBER
    %token <str> POWER
    ```

  - some specifications that certain operations are left-associative:

    ```
    %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 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.

[4]These functions for computing with quaternions 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 expressions involving quaternions.

- 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 happened to be sitting in the same directory.

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

Problem 1. [4 marks]

Construct `prob1.l`, as described above, so that it can be used with `plus-times-power.y` to build a parser for PLUS-TIMES-POWER.

Quaternion expressions

The *quaternions* are a system of four-dimensional numbers that are used in computer graphics to describe rotations in three-dimensional space, beginning with *Tomb Raider* in 1996 [1]. They were discovered by William Rowan Hamilton in Dublin in 1843.

Every quaternion has the form

$$w + xi + yj + zk,$$

where $w, x, y, z \in \mathbb{R}$ and i, j, k are special quantities, called *quaternion units*, satisfying

$$\begin{aligned} i^2 &= -1, \\ j^2 &= -1, \\ k^2 &= -1, \\ ijk &= -1. \end{aligned}$$

Other properties that follow from these equations include:

$$\begin{aligned} ij &= k, & ji &= -k, \\ jk &= i, & kj &= -i, \\ ki &= j, & ik &= -j. \end{aligned}$$

These properties can be used to compute any sum or product of quaternions, since the usual associative and distributive laws still apply. Note, though, that multiplication of quaternions is not commutative: the order of multiplication affects the outcome, in general.

You may have already met the *complex numbers*. These have the form $x + yi$, where $x, y \in \mathbb{R}$ and $i^2 = -1$, and may be used to describe rotations in two-dimensional space. It is an intriguing mathematical fact that, in order to extend complex numbers to describe rotations in three-dimensional space, a *four*-dimensional system is needed.

The set of quaternions is denoted by $\mathbb{H}$, just as the sets of real and complex numbers are denoted by $\mathbb{R}$ and $\mathbb{C}$ respectively.

In this assignment, you will construct a **quaternionic calculator** for parsing and evaluating arithmetic expressions involving quaternions.

We assume that, when representing the quaternion $w + xi + yj + zk$, the numbers w, x, y, z are represented in decimal form. A decimal point is optional for integers; if the fractional part is zero, then the decimal point may be absent, or it may be present but with no decimal digits after it, or it may be present with some number of 0s after it. For non-integers lying strictly between -1 and 1, the 0 before the decimal point may be present or absent. But every number with a decimal point must have something before the decimal point, or after the decimal point, or both. Trailing zeros are always allowed. So the number $-3/4$ may be represented as any of -0.75, -0.750, -0.7500, $-.75$, $-.750$, etc, and all these possibilities must be accepted as valid representations of the same real number.

We allow all the four arithmetic operations on quaternions — denoted by the usual symbols, `+`, `-`, `*`, `/` — and grouping by parentheses, `(...)`. We also allow the special operation `Rotation(...,...)`, written as a function, which creates a quaternion that represents a 3D rotation. The `Rotation` function takes two arguments, the first being a number, and the second being any quaternion expression.

The functions to do these operations have all been written for you and provided in the file `plus-times-power.y`. You only need to modify a copy of that file, using the guidance below, to build your calculator.

These operations can be combined in the same way in which they are combined when used for "normal" numbers (real, complex, etc.). Any valid expression can be given as the second argument of `Rotation(...,...)`, to give another valid expression, and expressions using `Rotation(...,...)` can be combined using arithmetic operations.

We formalise the concept of a *quaternion expression* with the following inductive definition:

1. Each of i, j, k is a quaternion expression.

2. If r is any nonnegative real number, then each of r, ri, rj, rk is a quaternion expression.

3. If p and q are quaternion expressions, then so are: (q), $\quad -q$, $\quad p+q$, $\quad p-q$, $\quad p*q$, $\quad p/q$.

4. If r is a nonnegative real number and q is a quaternion expression, then the following is a quaternion expression: $\text{Rotation}(r, q)$

Notes:

- Negative numbers can be represented by negating positive numbers.

- We allow juxtaposition of a real number r with any of i, j, k to form the simple expressions ri, rj and rk. This enables the succinct quaternion notation $\pm w \pm xi \pm yj \pm zk$. However, apart from that, multiplication in our quaternion expressions is always denoted by a star, $*$.

Let QUAT be the language of valid quaternion expressions in which all numbers are finite decimal representations. Here are some examples of valid quaternion expressions (i.e., members of QUAT):

expression	evaluates to
$(0.5 - 1.618j - 32i * j - k/(3.5 + i * j))/(i/j - k * 48)$	$0.658452 + -0.033020i + 0.000000j + 0.008664k$
$3i * j * k$	-3
$-1 + 2i - .3 * j + 4. * k$	$-1 + 2i - 0.3j + 4k$
0	0
$\texttt{Rotation}(120, 2i + 2j + 2k) * i/\texttt{Rotation}(120, 2i + 2j + 2k)$	j

Some examples of *invalid* quaternion expressions (i.e., not members of QUAT):

expression	comment
$3i * j\, k$	The product of j and k should use $*$ instead of juxtaposition.
$(-0.5 + 0.866j)^3$	Exponentiation is not possible in QUAT.
$\texttt{Power}(-0.5 + 0.866j, 3)$	`Power` is not valid in QUAT; we only use it in PLUS-TIMES-POWER.

We first treat QUAT as a language over the twelve-symbol alphabet {`i`, `j`, `k`, `+`, `-`, `*`, `/`, (,), `NUMBER`, `ROTATION`, `,` }.[5] Here, `NUMBER` is a token representing any finite decimal representation of a nonnegative real number, and `ROTATION` is a token representing the name of the Rotation function.

Problem 2. [2 marks]

Write a regular expression, using the regular expression syntax used by `lex`, that matches any finite decimal representation of a nonnegative real number.

Problem 3. [7 marks]

Write a Context-Free Grammar for the language QUAT over the twelve-symbol alphabet {`i`, `j`, `k`, `+`, `-`, `*`, `/`, (,), `NUMBER`, `ROTATION`, `,` }.

Now we use regular expressions (in the `lex` file, `prob4.l`) and a grammar (in the `yacc` file, `prob5.y`) to construct a lexical analyser (Problem 4) and a parser (Problem 5) for QUAT.

Problem 4. [7 marks]

Using the file provided for PLUS-TIMES-POWER as a starting point, construct a `lex` file, `prob4.l`, and use it to build a lexical analyser for QUAT.

Sample output:

```
$ ./a.out
Rotation(120,2i+2j+2k) * i / Rotation(120,2i+2j+2k)
Token: ROTATION;  Lexeme: Rotation
Token and Lexeme: (
Token: NUMBER;  Lexeme: 120
Token and Lexeme: ,
Token: NUMBER;  Lexeme: 2
Token and Lexeme: i
Token and Lexeme: +
```

[5] The last symbol listed in this set is a comma.

```
Token: NUMBER;  Lexeme: 2
Token and Lexeme: j
Token and Lexeme: +
Token: NUMBER;  Lexeme: 2
Token and Lexeme: k
Token and Lexeme: )
Token and Lexeme: *
Token and Lexeme: i
Token and Lexeme: /
Token: ROTATION;  Lexeme: Rotation
Token and Lexeme: (
Token: NUMBER;  Lexeme: 120
Token and Lexeme: ,
Token: NUMBER;  Lexeme: 2
Token and Lexeme: i
Token and Lexeme: +
Token: NUMBER;  Lexeme: 2
Token and Lexeme: j
Token and Lexeme: +
Token: NUMBER;  Lexeme: 2
Token and Lexeme: k
Token and Lexeme: )
Token and Lexeme: <newline>
```
Control-D

Problem 5. [9 marks]

Make a copy of `prob4.l`, call it `prob5.l`, then modify it so that it can be used with `yacc`. Then construct a `yacc` file `prob5.y` from `plus-times-power.y`. Then use these `lex` and `yacc` files to build a parser for QUAT.

Note that you do not have to program any of the quaternion functions yourself. They have already been written: see the Programs section of the `yacc` file. The *actions* in your `yacc` file will need to call these functions, and you can do that by using the function call for `pow(...)` in `plus-times-power.y` as a template.

The core of your task is to write the grammar rules in the Rules section, in `yacc` format, with associated actions, using the examples in `plus-times-power.y` as a guide. You also need to do some modifications in the Declarations section; see page 5, and further details below.

When entering your grammar into the Rules section of `prob5.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 `ROTATION` token. It has the same structure as the line for the `NUMBER` token, except that "`num`" is replaced by "`str`" (since `ROTATION` represents a string, being a name for a function, whereas `NUMBER` represents a number.
- For symbols that represent a binary (i.e., two-argument) arithmetic operation, it is worth including them in an appropriate `%left` statement. Each of these statements makes the parser treat these operations as left-associative, which helps it determine the order in which to do the operations and removes some sources of possible ambiguity. When using `%left`, operations

having the same precedence are listed on the same line with spaces between them. So for `+` and `-` you can use the following statement:

```
%left '-' '+'
```

A similar line can be used for multiplication and division. For operations whose `%left` statements are on different lines, the operations with higher precedence are those with higher line numbers (i.e., later in the file). (Right-associative operations can be handled similarly with a `%right` statement, though we don't have any such operations here.)

- For every nonterminal symbol, you need 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. For example,

  ```
  %type <qtn> start
  ```

  Here, "`qtn`" is the type name we are using for quaternions. 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
Rotation(120,2i+2j+2k) * i / Rotation(120,2i+2j+2k)
0.000000 + 0.000000 i + 1.000000 j + 0.000000 k
Control-D
```

Now let's apply quaternion calculations to rotate a point around an axis in 3D space. Throughout, we assume the axis goes through the origin.

Points in 3D space are represented as *pure quaternions*, which means quaternions of the form $xi + yj + zk$, so they have no real part w. This is just another way of representing three-dimensional space using standard co-ordinate axes. In effect, the three basic quaternion units i, j, k are unit vectors along the x, y, z-axes, respectively, and correspond to points $(1,0,0), (0,1,0), (0,0,1)$ respectively.[6]

A rotation is specified by giving its axis as a unit vector, i.e., a pure quaternion of length 1, and its angle as a real number. If the unit-length pure quaternion $\hat{q}$ gives the direction of the axis of rotation, and θ is the angle of rotation around that axis (clockwise, as viewed from the origin looking in the direction in which q points), then the quaternion $\texttt{Rotation}(\theta, \hat{q})$ that represents the rotation is given by

$$\texttt{Rotation}(\theta, \hat{q}) \quad = \quad \cos(\theta/2) + \sin(\theta/2) \cdot \hat{q}.$$

It is this quaternion that is returned by the function `rotation` (provided in `plus-times-power.y`) and by the `Rotation` operation in quaternion expressions.

In order to apply the rotation to a point p, we first represent p as a pure quaternion, $p = xi + yj + zk$, and then form the product

$$\texttt{Rotation}(\theta, \hat{q}) * p \,/\, \texttt{Rotation}(\theta, \hat{q}).$$

So, our earlier calculation that `Rotation(120,2i+2j+2k) * i / Rotation(120,2i+2j+2k)` evaluates to `j` expresses the fact that, if your axis is the line with direction $i + j + k$, then rotating the point $(1,0,0)$ by $120°$ clockwise around this axis gives the point $(0,1,0)$.

In this assignment, we restrict to angles $\theta \geq 0$. We lose no generality by doing this, since any rotation is equivalent to rotation about the same axis by some angle θ in the range $0° \leq \theta < 360°$.

[6]BUT the multiplication we use for quaternions is not the same as the dot product or cross product of vectors, although it is closely related to both. In fact, historically, quaternions gave rise to these vector products.

Problem 6. [5 marks]

Convert your eight-digit student ID number into an angle and direction as follows. Let

$$d_1d_2d_3d_4d_5d_6d_7d_8$$

be the digits of your student ID number. Divide this into one two-digit number followed by six single-digit numbers: d_1d_2, d_3, d_4, d_5, d_6, d_7, d_8. The first of these, d_1d_2, is your angle of rotation, in degrees. The axis of rotation is the line whose direction is given by $d_3i + d_4j + d_5k$. The point to be rotated is (d_6, d_7, d_8), which can be represented by the pure quaternion $d_6i + d_7j + d_8k$.

(a) Write down the quaternion expression in QUAT that represents the calculation required to rotate the point (d_6, d_7, d_8) by d_1d_2 degrees clockwise around the axis whose direction is given by $d_3i + d_4j + d_5k$.

(Your expression must use the actual digits derived from your student ID number, not the algebraic quantities used above.)

(b) Run your parser on your expression from (a), and report the result of evaluating it.

The answers to (a) and (b) should be copied into a single line each in the file `prob6.txt`.

Turing machines

A *signed quaternion unit product* is a string over the alphabet {`+`, `-`, `i`, `j`, `k`} which has an optional sign, `+` or `-`, at the start, at least one character as well as any initial sign, and all other characters are `i` or `j` or `k`. We interpret such a string as a product of quaternions, each from the set $\{i, j, k\}$, possibly negated. Our next task is to write a Turing machine to evaluate products of this type.

The following table shows some examples of signed quaternion unit products and, for each, the result of evaluating it.

input	output
`ijk`	`-1`
`j`	`j`
`-k`	`-k`
`+ii`	`-1`
`-ii`	`1`
`-iijikkjikjikijj`	`-j`

If the result is one of $1, i, j, k$, then we omit the sign, so `+` does not appear at the start of the output, even though it is allowed, optionally, at the start of the input string. So the output string must be one of the following eight possibilities:

`1`, `-1`, `i`, `-i`, `j`, `-j`, `k`, `-k`.

Problem 7. [8 marks]

Build, in Tuatara, a Turing machine to evaluate any signed quaternion unit product.

When your Turing machine stops at the end of the computation,

- the tape head must be at the first square of the tape, and
- all tape cells beyond the output string must be blank.

Since Tuatara tape alphabets can only contain alphanumeric characters, use `P` for `+` and `M` for `-`.

Context-Free Languages

This question is about the language CricketBowlingFigures defined in Tutorial 4.

Problem 8. [8 marks]

Prove or disprove: the language CricketBowlingFigures is context-free.

References

[1] Nick Bobic, Rotating objects using quaternions, *Game Developer*, Feb. 1998. Available at: `https://www.gamasutra.com/view/feature/3278/rotating_objects_using_quaternions.php?print=1`

[2] Daniel Chan, Quaternions are turning tomb raiders on their heads, *Parabola* **40** (no. 2) (2004). Available at: `https://www.parabola.unsw.edu.au/files/articles/2000-2009/volume-40-2004/issue-2/vol40_no2_2.pdf` or `https://web.maths.unsw.edu.au/~danielch/talent/quat1.pdf`