Monash University
Faculty of Information Technology
2^{nd} Semester 2019

FIT2014
Assignment 2
Regular Languages, Context-Free Languages, Pushdown Automata, Lexical analysis, Parsing, and Turing machines
DUE: 11:55pm, Friday 11 October 2019

In these exercises, you will

- implement lexical analysers using `lex` (Problem 3);
- implement parsers using `lex` and `yacc` (Problems 1, 4);
- practise proof by induction (Problem 2);
- practise using the Pumping Lemma for regular languages (Problem 7);
- learn some more about Turing machines (Problem 5);
- learn more about Pushdown Automata (Problems 6–7).

How to manage this assignment

- You should start working on this assignment now, and spread the work over the time until it is due. Aim to do at least three questions before the mid-semester break. Do as much as possible *before* your week 10 prac class. There will not be time during the class itself 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 need to construct new `lex` files, using `chain.l` as a starting point, for Problems 1, 3 & 4, and you'll need to construct a new `yacc` file from `chain.y` for Problem 4. Your submission must include (as well as the appropriate PDF files for the exercises requiring written solutions):

- a `lex` file `prob1.l` which should be obtained by modifying a copy of `chain.l`
- a `lex` file `prob3.l` which should also be obtained by modifying a copy of `chain.l`
- a `lex` file `prob4.l` which should be obtained by modifying a copy of `prob3.l`
- a `yacc` file `prob4.y` which should be obtained by modifying a copy of `chain.y`
- PDF files for the exercises requiring written solutions, namely, `prob1.pdf`, `prob2.pdf`, `prob5.pdf`, `prob6.pdf`, and `prob7.pdf`.

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 CHAIN 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 CHAIN based on certain expressions involving strings. Then you will use `lex` and `yacc` on a language CRYPT of expressions based on cryptographic operations.

CHAIN

The language CHAIN consists of expressions of the following type. An expression consists of a number of terms, with `#` between each pair of consecutive terms, where each term is either a string of lower-case letters or an application of the `Reverse` function to such a string. Examples of such expressions include

```
mala # y # Reverse(mala)
block # drive # cut # pull # hook # sweep # Reverse(sweep)
Reverse(side) # Reverse(direction) # Reverse(gear)
```

For lexical analysis, we wish to treat every lower-case alphabetical string as a lexeme for the token STRING, and the word `Reverse` as a lexeme for the token REVERSE.

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, `chain.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

pattern { *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 string of lower-case letters;
  - This is taken to be an instance of the token STRING (i.e., a lexeme for that token).
- the specific string `Reverse`;
  - Such a string is taken to be an instance of the token REVERSE.
- certain specific characters: `#`, `(`, `)`;
- white space, being any sequence of spaces and tabs;
- the newline character.

Note that all matching 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 `chain.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 chain.l
$ cc lex.yy.c
```

By default, `cc` puts the executable program in a file called `a.out`.[3] 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 `chain-lex`, then you can tell this to the compiler using the `-o` option: `cc lex.yy.c -o chain-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 `chain.l`. Note how it skips over white space, and only reports on the lexemes and tokens.

```
$ ./a.out
mala     #  y #Reverse( mala)
Token: STRING;  Lexeme: mala
Token and Lexeme: #
Token: STRING;  Lexeme: y
Token and Lexeme: #
Token: REVERSE;  Lexeme: Reverse
Token and Lexeme: (
Token: STRING;  Lexeme: mala
Token and Lexeme: )
Token and Lexeme: <newline>
```

[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.

Try running this program with some input expressions of your own.

Yacc

We now turn to parsing, using `yacc`.

Consider the following grammar for CHAIN.

$$\begin{array}{rcl} S & \longrightarrow & E \\ S & \longrightarrow & \varepsilon \\ E & \longrightarrow & E\#E \\ E & \longrightarrow & \textbf{STRING} \\ E & \longrightarrow & \textbf{REVERSE}(\textbf{STRING}) \end{array}$$

In this grammar, the non-terminals are S and E. Treat **STRING** and **REVERSE** 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 concatenation and `Reverse(...)` interpreted as reversing a string.

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

- Copy `chain.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 = ...;`
      - `return` *TOKENNAME*`;`
      - `return *yytext;`
    - 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 about four lines at the end of the file.
- `chain.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 `chain.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

    ```
    char *reverse(char *);
    char *simpleSub(char *, char*);
    ⋮
    ```

    which are *declarations* of functions (but they are *defined* later, in the Programs section);

– declarations of the tokens to be used:

```
%token <str> STRING
%token <str> REVERSE
```

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

```
%type <str> start
%type <str> expr
```

– 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.
- 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 chain.y
$ flex prob1.l
$ cc lex.yy.c y.tab.c
```

The executable program, which is now a parser for CHAIN, 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
mala     #  y #Reverse( mala)
malayalam
```
[4]

Run it with some input expressions of your own.

Problem 1. [7 marks]

(a) Construct `prob1.l`, as described above, so that it can be used with `chain.y` to build a parser for CHAIN.

(b) Show that the grammar for CHAIN given above is ambiguous.

[4] Malayalam is the main language of the southern Indian state of Kerala. The word was given as an example of a palindrome by an FIT2014 student in a lecture in 2017.

(c) Find an equivalent grammar (i.e., one that generates the same language) that is not ambiguous.

Cryptographic expressions

A **cryptographic calculator** performs simple operations on strings of a kind that are used in classical cryptosystems. It is also able to combine these operations in a natural way.

Suppose $x = x_1 x_2 \cdots x_n$ and $k = k_1 k_2 \cdots k_t$ are two strings, where x_i and k_i denote the i-th letters of x and k respectively.

The available operations are:

- **sum**: this is written $x + k$ in our expressions, though our C function that computes it is called `sum(...)`. The resulting string has length $\min\{n, t\}$, and its i-th letter is $x_i + k_i \bmod 26$, where the letters of the English alphabet correspond to numbers via `a` $= 0$, `b` $= 1$, ..., `z` $= 25$.

  For example,

  $x =$ `thebushwasalivewithexcitement`
  $k =$ `mrskoalahadabrandnewbabyandthenewsspreadlikewildfire`
  $x + k =$ `fywlisswhsdljmejlglaycjrezhga`

- **difference**: this is written $x - k$, and is computed by the C function `diff(...)`. The resulting string has length $\min\{n, t\}$, and its i-th letter is $x_i - k_i \bmod 26$.

- **Vigenère cypher**: this is written `Vigenere`(x, k), where x is the *plaintext* and k is the *key*. We first concatenate k with itself as many times as necessary in order to make it at least as long as x. Then we form the sum. The result is a string whose i-th letter is

  $$(x_i + k_{((i-1) \bmod t)+1}) \bmod 26.$$

  For example, if the plaintext is

  `inaholeinthegroundtherelivedahobbit`

  and the key is

  `bilbo`

  then `Vigenere(inaholeinthegroundtherelivedahobbit, bilbo)` returns the cyphertext

  `jvlicmmtohimrscvvouvfzpmwwmobvpjmjh`

- **Simple Substitution:** this is written `SimpleSub`(x, k). Again, x is plaintext, and k is the key, but this time k must be a permutation of the 26-letter English alphabet, represented as a string in which each letter appears exactly once (and $t = 26$). Every a in the plaintext is replaced by the 1st letter k_1 of the key; every `b` in the plaintext is replaced by the 2nd letter, k_2, of the key; and so on. In general, the plaintext letter x_i is replaced by k_{x_i}.

  For example, if the plaintext is

  `thequickbrownfoxjumpsoverthelazydog`

  and the key is

  `qwertyuiopasdfghjklzxcvbnm`

  then `SimpleSub(thequickbrownfoxjumpsoverthelazydog,qwertyuiopasdfghjklzxcvbnm)` returns the cyphertext

  `zitjxoeawkgvfygbpxdhlgctkzitsqmnrgu`

- **Local Transposition:** this is written $\texttt{LocTran}(x,k)$. The letters of the plaintext x are not replaced, as happens in the previous cyphers, but rather rearranged according to a permutation, which is represented by a string k of w digits, where $w = |k|$.[5] This permutation string k has length ≤ 10, and consists of the digits $0, 1, \ldots, w-1$ arranged in some order. For example, if k has length 3 (so $w = 3$), then k can be any of the strings 012, 021, 102, 120, 201, 210. The plaintext x is divided into blocks of w letters each, and the letters within each block are permuted according to k. If there are extra letters at the end — too few letters to make up another full block — then these are just copied across, with no change in their positions.

  For example, if the plaintext is

  `thefamilyofdashwood`

  and the key is

  `201`

  then `LocTran(thefamilyofdashwood,201)` returns the cyphertext

  `ethmfayildofhasowod`

These operations can be combined. Any valid expression can be given as the first argument to one of the cypher functions, or as any argument of a sum or difference, to give another valid espression. So you can form expressions like

`Vigenere(LocTran(triantiwontigongolope,3201),bunyip) + muldjewangk`

Let CRYPT be the language of cryptographic expressions of this type that can be generated by the following grammar.

$$
\begin{array}{lcl}
S & \longrightarrow & E \\
S & \longrightarrow & \varepsilon \\
E & \longrightarrow & E + E \\
E & \longrightarrow & E - E \\
E & \longrightarrow & (E) \\
E & \longrightarrow & \texttt{SIMPLESUB}(E, \texttt{STRING}) \\
E & \longrightarrow & \texttt{VIGENERE}(E, \texttt{STRING}) \\
E & \longrightarrow & \texttt{LOCTRAN}(E, \texttt{DIGITS}) \\
E & \longrightarrow & \texttt{STRING}
\end{array}
$$

In this grammar, the non-terminals are S and E. Treat `SIMPLESUB`, `VIGENERE`, `LOCTRAN`, `STRING` and `DIGITS` as just single tokens. For `SIMPLESUB`, `VIGENERE`, and `LOCTRAN`, we allow any nonempty prefix of the function name as well as the full name; e.g., `S`, `Si`, `Sim`, …, `SimpleSu`, `SimpleSub` are all acceptable lexemes for the token SIMPLESUB.

Problem 2. [7 marks]

For each $n \geq 0$, let V_n be the string

$$\underbrace{\texttt{VIGENERE(}\cdots\texttt{VIGENERE(VIGENERE(}}_{n\text{ times}}\texttt{STRING}\underbrace{\texttt{,STRING),STRING)}\cdots\texttt{,STRING)}}_{n\text{ times}}$$

where the Vigenère cypher is applied n times.

Prove, by induction on n, that V_n has a derivation, using the above grammar, of length $n+2$.

[5] w stands for $w$idth, the traditional term for the size of a permutation used in local transposition.

Problem 3. [7 marks]

Using the file provided for CHAIN as a starting point, construct a `lex` file, `prob3`, and use it to build a lexical analyser for CRYPT.

Sample output:

```
$ ./a.out
Loc(Sim(Vig(therewasmovementatthestation,banjo),thequickbrownfxjmpsvlazydg),10)
Token: LOCTRAN;  Lexeme: Loc
Token and Lexeme: (
Token: SIMPLESUB;  Lexeme: Sim
Token and Lexeme: (
Token: VIGENERE;  Lexeme: Vig
Token and Lexeme: (
Token: STRING;  Lexeme: therewasmovementatthestation
Token and Lexeme: ,
Token: STRING;  Lexeme: banjo
Token and Lexeme: )
Token and Lexeme: ,
Token: STRING;  Lexeme: thequickbrownfxjmpsvlazydg
Token and Lexeme: )
Token and Lexeme: ,
Token: DIGITS;  Lexeme: 10
Token and Lexeme: )
Token and Lexeme: <newline>
Control-D
$ ./a.out
V(twentysix - eleven, eleven)
Token: VIGENERE;  Lexeme: V
Token and Lexeme: (
Token: STRING;  Lexeme: twentysix
Token and Lexeme: -
Token: STRING;  Lexeme: eleven
Token and Lexeme: ,
Token: STRING;  Lexeme: eleven
Token and Lexeme: )
Token and Lexeme: <newline>
```

Problem 4. [7 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 `chain.y`. Then use these `lex` and `yacc` files to build a parser for CRYPT.

Note that you do not have to program any of the cryptographic 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 `reverse(...)` in `chain.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 `chain.y` as a guide. You also need to do some modifications in the Declarations section to declare all tokens, using `%token`, and declare all nonterminal symbols, using `%type`.[a] See page 5.

[a]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
Loc(Sim(Vig(therewasmovementatthestation,banjo),thequickbrownfxjmpsvlazydg),10)
kltpysiteauzfglhctaesircrktx
Control-D
$ ./a.out
V(twentysix - eleven, eleven)
twenty
```

Turing machines

Problem 5. [3 marks]

A *Forgetful Turing Machine* (FTM) operates just like a normal Turing machine except that, in every instruction (i.e., transition), the letter *written* in the tape cell is *always* the letter ‘`a`’, regardless of the current state and the current letter (although the read/write head is still allowed to move either Left or Right, according to the instruction).

What class of languages is recognised by FTMs? Justify your answer.

The language of encoded Pushdown Automata

In the next two questions, your Pushdown Automata use input alphabet {`a`,`b`} and stack alphabet {`a`,`b`,`$`}. Their start state is always state 1.

Problem 6 should help prepare for Problem 7.

CWL-PDA: the Code-Word Language for Pushdown Automata

In Lecture 15, we learned how to encode Turing machines as strings in CWL, the Code-Word Language. We now describe a similar method for encoding Pushdown Automata which will be used in Problems 6 and 7.

Every PDA (of the above type) can be encoded as a string over the alphabet {`a`,`b`} as follows.

We assume that the states are designated by positive integers. There is no requirement to use consecutive numbers; for example, it's ok to have a three-state PDA with state numbers 1, 4 and 1966.

For each n, a state numbered n is denoted by the string $\mathtt{a}^n\mathtt{b}$.

In specifying transitions, symbols are encoded according to the following table:

symbol s	code $\langle s \rangle$
`a`	`aa`
`b`	`ab`
`$`	`ba`
ε	`bb`

Recall that a PDA transition has the form

$$\textcircled{m} \xrightarrow{x,\ y \to z} \textcircled{n}$$

where

- m and n are states
- x is an input alphabet symbol or ε,
- y, z are stack alphabet symbols or ε.

We encode such a transition as the string

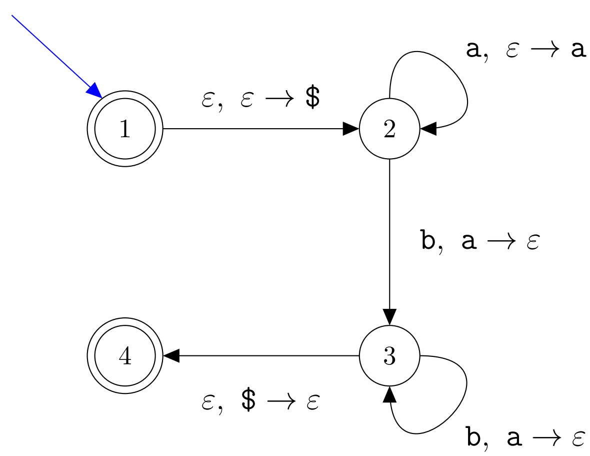
$$\mathtt{a}^m\mathtt{ba}^n\mathtt{b}\,\langle x\rangle\,\langle y\rangle\,\langle z\rangle$$

where the symbols x, y, z are encoded as $\langle x\rangle$, $\langle y\rangle$, $\langle z\rangle$ according to the above table. (Note that the angle-brackets $\langle$ and $\rangle$ do not appear in the string! We are just using them to denote the act of encoding.)

The entire PDA is then encoded by

1. converting each transition into a string as above, and then concatenating all those strings;
2. appending the string bbbbbbbb, as a delimiter (note that this string cannot occur previously in our encoding so far, so we are using it to mark the end of the listing of all the transitions);
3. for each final state f, append the string $\mathtt{a}^f\mathtt{b}$.

For example, consider the PDA introduced in Lecture 11 that recognises the language $\{\mathtt{a}^i\mathtt{b}^i : i \geq 0\}$:



The following table lists its transitions and the strings that encode them.

From state	To state	x	y	z	string
1	2	ε	ε	\$	abaabbbbbba
2	2	a	ε	a	aabaabaabbaa
2	3	b	a	ε	aabaaababaabb
3	3	b	a	ε	aaabaaababaabb
3	4	ε	\$	ε	aaabaaaabbbbabb

So the string that encodes this PDA is:

$$\underbrace{\mathtt{abaabbbbbba}}_{1\longrightarrow 2}\underbrace{\mathtt{aabaabaabbaa}}_{2\longrightarrow 2}\underbrace{\mathtt{aabaaababaabb}}_{2\longrightarrow 3}\underbrace{\mathtt{aaabaaababaabb}}_{3\longrightarrow 3}\underbrace{\mathtt{aaabaaaabbbbabb}}_{3\longrightarrow 4}\underbrace{\mathtt{bbbbbbbb}}_{\text{delimiter}}\underbrace{\mathtt{abaaaab}}_{\text{final states}}$$

CWL-PDA denotes the language of encodings, according to the above scheme, of all PDAs.

We say that a string in CWL-PDA is *valid* if at least one of the transitions encoded in it includes State 1. (For validity, it's enough for State 1 to appear as either a From-state or a To-state. But if it only appears as a To-state then the PDA will always crash, regardless of the input; it will be useless, even though it's valid.)

Observe that CWL-PDA has many strings that represent PDAs that are either invalid or useless.

Problem 6. [11 marks]

(a) Prove that the language CWL-PDA is regular, by giving a regular expression for it.

(b) Write, in table form, a three-state PDA that accepts the input string `ab` and *crashes or rejects* for *every other input string.*

(c) On a single line, write down a string over {`a`,`b`} that represents your PDA from part (b) in the language CWL-PDA.

(d) How many different strings in CWL-PDA represent this *exact same* PDA?

- These strings must all represent *identical* PDAs, not just PDAs that are equivalent in the sense that they always give the same result.
- You may assume that strings in CWL-PDA represent each PDA table row exactly once.

(e) Explain how to construct, for each $n \geq 2$, a three-state PDA M_n that accepts the input string `ab`, *crashes or rejects* for *every other input string*, and whose CWL-PDA encoding starts with $\mathtt{a}^n$.

Let AB-PDA be the language of all strings in CWL-PDA that represent PDAs that (i) are valid, and (ii) accept the input `ab`. For example, if you have done part (c) correctly, the string you constructed there should belong to AB-PDA.

(f) Give a string in AB-PDA that has the property that, if at least one '`a`' is inserted at the very start of the string, the resulting string is still valid but is no longer in AB-PDA.

Problem 7. [8 marks]

Use the Pumping Lemma for Regular Languages to show that AB-PDA is not regular.

FOR BONUS MARKS: determine, with proof, whether or not AB-PDA is decidable.

References

- Jane Austen, *Sense and Sensibility*, Thomas Egerton, London, 1811.
- C. J. Dennis, *The Triantiwontigongolope*, poem in his book, *A Book for Kids*, Angus & Robertson, Sydney, 1921.
- A. B. 'Banjo' Patterson, *The Man from Snowy River*, poem first published in *The Bulletin* on 26 April 1890.
- J. R. R. Tolkien, *The Hobbit*, Allen & Unwin, London, 1937.
- Dorothy Wall, *Blinky Bill*, Angus & Robertson, Sydney, 1933.