Lab 7

COMP9021, Session 1, 2014

The aim of this lab is to practice the use of pointers and command line arguments, and provide an introduction to context free grammars.

1 ☞ Splitting a sequence into 4 classes

Complement the following program, whose aim is to finds all possible ways of inserting 3 multiplication signs in the sequence 1238965740 in two different manners such that both resulting products of 4 numbers are equal and not equal to 0.

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

bool create_partition(int *const, int *const, int *const);
void generate_numbers(const int, const int, const int, int *const);
void print_solution(const int *const, const int *const, const int);

const int digits[] = {1, 2, 3, 8, 9, 6, 5, 7, 4, 0};

int main(void) {
    int i1 = 1, j1 = 2, k1 = 2;
    /* Give i1, j1 and k1 all possible values with 0 < i1 < j1 < k1 < 9. */
    while (create_partition(&i1, &j1, &k1)) {
        int numbers1[4];
        /* Store in numbers1 the number consisting
         * of the first i1 digits in digits[],
         * then the number consisting of the j1 - i1 next digits in digits[],
         * then the number consisting of the k1 - j1 next digits in digits[],
         * and eventually the number consisting
         * of the remaining digits in digits[],
         * with at least 2 of those remaining digits.*/
        generate_numbers(i1, j1, k1, numbers1);
        int i2 = i1;
        int j2 = j1;
        int k2 = k1;
        /* Give i2, j2 and k2 all possible values with 0 < i2 < j2 < k2 < 9
         * and (i1, j1, k1) < (i2, j2, k2). */
        while (create_partition(&i2, &j2, &k2)) {
            int numbers2[4];
            /* Store in numbers2 the number consisting
```

```
             * of the first i2 digits in digits[],
             * then the number consisting of the j2 - i2 next digits in digits[],
             * then the number consisting of the k2 - j2 next digits in digits[],
             * and eventually the number consisting
             * of the remaining digits in digits[],
             * with at least 2 of those remaining digits.*/
            generate_numbers(i2, j2, k2, numbers2);
            int product = numbers1[0] * numbers1[1] * numbers1[2] * numbers1[3];
            if (numbers2[0] * numbers2[1] * numbers2[2] * numbers2[3] == product)
                print_solution(numbers1, numbers2, product);
        }
    }
    return EXIT_SUCCESS;
}
```

2 Syntactically correct arithmetical expressions

A *context free* grammar is a set of *production rules* of the form

`symbol_0 ---> symbol_1 ... symbol_n`

where `symbol_0 ...symbol_n` are either *terminal* or *nonterminal symbols*, with `symbol_0` being necessarily nonterminal. A symbol is a nonterminal symbol iff it is denoted by a word built from underscores or uppercase letters. A special nonterminal symbol is called the *start symbol*. The language *generated* by the grammar is the set of sequences of terminal symbols obtained by replacing a nonterminal symbol by the sequence on the right hand side of a rule having that nonterminal symbol on the left hand side, starting with the start symbol. For instance, the following, where `EXPRESSION` is the start symbol, is a context free grammar for a set of arithmetic expressions.

```
EXPRESSION --> TERM SUM_OPERATOR EXPRESSION
EXPRESSION --> TERM
TERM --> FACTOR MULT_OPERATOR TERM
TERM --> FACTOR
FACTOR --> NUMBER
FACTOR --> (EXPRESSION)
NUMBER --> DIGIT NUMBER
NUMBER --> DIGIT
DIGIT --> 0
...
DIGIT --> 9
SUM_OPERATOR --> +
SUM_OPERATOR --> -
MULT_OPERATOR --> *
MULT_OPERATOR --> /
```

Moreover, blank characters (spaces or tabs) can be inserted anywhere except inside a number. For instance, `(2 + 3) * (10 - 2) - 12 * (1000 + 15)` is an arithmetic expression generated by the grammar.

Verify that the grammar is *unambiguous*, in the sense that every expression generated by the grammar has a unique evaluation.

Write down a program that prompts for an expression to be entered over one line (ended by the first `'\n'` character), checks whether it can be generated by the grammar, and in case the answer is yes, evaluates the expression. Then the program checks for a new expression, and stops in case the user enters an empty line, following this kind of interaction:

```
$ a.out
Enter an expression: 2 * 2
Syntax is correct
Value is 4
Enter an expression: (2 + 3) * (10 - 2) - 12 * (1000 + 15)
Syntax is correct
Value is -12140
Enter an expression:  2 ++ 3
Syntax is incorrect
Enter an expression:
Bye
```

You might find the function `ungetc()`, declared in the header file `<stdio.h>`, especially useful. It returns an `int` and takes two inputs: a `char` and for our purposes, `stdin`. Its effect is to make as if the first input had been pushed back to standard input, being ready to be read again (by `getchar()` for instance).

3 ☞ Simulating a Unix utility with options

The `tr` Unix utility is used to **tr**anslate characters. It copies the text coming from standard input to standard output with substitution, deletion or compression of selected characters. In the following, we use `echo` to output some text that we pipe with the `|` symbol to the standard input of the next command (either `tr` or `a.out`).

- To substitute characters, `tr` takes two strings that we assume are of equal length as arguments, and replaces all occurrences of a character given in the first string by the corresponding character in the second string. For example:

  ```
  $ echo 'To tax or not to tax, that is the question.' | tr 'tax.' 'TAX!'
  To TAX or noT To TAX, ThAT is The quesTion!
  ```

- To delete characters, `tr` takes the `-d` option followed by a string, and deletes all occurrences of characters given in the string. For example:

```
$ echo 'rebels love rice' | tr -d 'br'
eels love ice
```

- To compress characters, `tr` takes the `-s` option followed by a string, and squeezes all consecutive occurrences of any character given in the string to one. For example:

```
$ echo 'Arghhhhhh!!!!!!! This gets on my neeeerves!!!!' | tr -s 'h!e'
Argh! This gets on my nerves!
```

- The `-s` option can also be used when substituting characters; the characters in the second string are then squeezed. For example:

```
$ echo 'To taxxxxxx or not to taxxxxx, that is the question...' | tr -s 'tax.' 'TAX!'
To TAX or noT To TAX, ThAT is The quesTion!
```

Write a program C program that implements the functionalities of `tr` just described, reporting "Syntax error" if appropriate. For example:

```
$ echo 'To tax or not to tax, that is the question.' | a.out 'tax.' 'TAX!'
To TAX or noT To TAX, ThAT is The quesTion!
$ echo 'rebels love rice' | a.out -d 'br'
eels love ice
$ echo 'Arghhhhhh!!!!!!! This gets on my neeeerves!!!!' | a.out -s 'h!e'
Argh! This gets on my nerves!
$ echo 'To taxxxxxx or not to taxxxxx, that is the question...' | a.out -s 'tax.' 'TAX!'
To TAX or noT To TAX, ThAT is The quesTion!
$ echo 'To tax or not to tax, that is the question.' | a.out 'tax' 'TAX!'
Syntax error
$ echo 'rebels love rice' | a.out 'br'
Syntax error
$ echo 'To tax or not to tax, that is the question.' | a.out -d 'tax.' 'TAX!'
Syntax error
```