ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecturer: Prof. B. Gärtner

Informatik für Mathematiker und Physiker HS13

Exercise Sheet 12

Submission deadline: 15:15 - Tuesday 10th December, 2013

Course URL: http://www.ti.inf.ethz.ch/ew/courses/Info1_13/

Assignment 0

This homework asks you to extend the class `ifmp::list`. Download the files `node.cpp`, `node.h`, `list.cpp`, `list.h` and `list_test.cpp` from the course website and store them in a folder together with the `Makefile` (the same file that we already used for the previous homeworks). The program `list_test.cpp` demonstrates how you can use the class `ifmp::list`. In extension to what you have seen in the lecture, `ifmp::list` can also be used with template arguments like `ifmp::list<int>` for a list of `int` and `ifmp::list<double>` for a list of `double` (like `std::vector<int>` and `std::vector<double>`.

Assignment 1 – Skript-Aufgabe 155 (4 points)

Implement a comparison operator for the class `ifmp::list`, in form of a global `operator==`. You can use the program `test_list.cpp` to test your operator.

```
// POST: returns true if and only if l1 and l2 store the same
//       sequence of keys
template <typename T>
bool operator==(const ifmp::list<T>& l1, const ifmp::list<T>& l2);
```

Assignment 2 (4 points)

Implement a member function `reverse()` for the class `ifmp::list`. You should implement `reverse()` so that the reversal is done *in-place*. This means that you do *not* create any temporary nodes or lists, but you do should modify the `next_` pointers of the nodes directly.

```
// POST: the elements of *this are in reverse order
template <typename T>
void list<T>::reverse();
```

Assignment 3 (4 points)

A *stacks* is a simple type data structure that stores a sequence of elements and works according to the LIFO principle (last-in first-out): A new element can be added only *at the front/top* of the stack. Only the first/top element can be accessed: you can retrieve the key of this element by calling `top()`, or you can remove the front/top element by calling `pop()`.

The file `stack.h` provides the definition of the class `ifmp::stack`. Implement this class and test it (you can use the program `stack_test.cpp`).

```
namespace ifmp {
  template <typename T>
  class stack {
   public:
    // Constructor
    stack();

    // POST: returns the value of the first element of *this
    // PRE:  *this is not empty
    T top() const;

    // POST: key was added before the first element of *this
    void push(T key);

    // POST: first element was removed from *this
    //       if *this is empty nothing happened
    void pop();

    // POST: returns true if and only if *this is empty
    bool empty();

   private:
    list<T> l_;
  }
}
```

Assignment 4 (4 points)

In Homework 9 you had to write several programs to draw turtle graphics for Lindenmayer systems. You wrote the production and drawing rules for each Lindenmayer system directly in the source code of the program. Therefore, you have to write a new program for every new Lindenmayer system you had to draw. Write a program `generic_lindenmayer.cpp` that lets you draw arbitrary Lindenmayer systems. The program should read from the input the number of iterations, the rotation angle (for the + and − symbol), the start word and the drawing- and production rules for each symbol. The symbols can be arbitrary letters, except +, − and `F`, which is the special symbol for "move one step forward". The rules could for instance be entered in the following format `c s1 s2`, where `c` is a character that describes the symbol, `s1` is a string

that describes the drawing rule for that symbol and `s2` is a string that describes the production rule. For instance `a F a+a` encodes the production rule $(a \to a + a)$ and the symbol a is drawn as F ("move one step forward"). Or `x # x-a` encodes the production rule $(x \to x - a)$, and nothing is drawn (any character different from `+-F` should simply be ignored). A sample run of your program could for instance look like this:

```
How many symbols =? 2
Enter symbol 1 drawing and production rule =? x # x+yF+
Enter symbol 2 drawing and production rule =? y # -Fx-y
Rotation angle =? 90
Number of iterations =? 14
Start word =? x
```

The dragon curve is drawn. Check carefully whether your program works correctly. Try to reproduce the dragon curve or the Sierpinski triangle (Exercise 3.a from Homework 9).