Informatik I (D-ITET)
Übungsstunde 11, 4.12.2017

Hossein Shafagh
shafagh@inf.ethz.ch
Problem 10.1. Recursive Function Analysis

(a) bool f (const int n) {
    if (n == 0) return false;
    return !f(n-1);
}

(b) void g (const int n) {
    if (n == 0) {
        std::cout << "*";
        return;
    }
    g(n-1);
    g(n-1);
    g(n-1);
}
Problem 10.1. Recursive Function Analysis

- Problem 10.1. Recursive Function Analysis

```cpp
// PRE: n>=0
// POST: return value is false if n is even and true if n is odd
bool f (const int n) {
    if (n == 0) return false;
    return !f(n-1);
}
```

- The function f immediately terminates for n == 0. With each recursive call n is decremented, i.e., f is called with parameter < n, eventually reaching n = 0.

- \( \text{Calls}_f(n) = n + 1 \)
Problem 10.1. Recursive Function Analysis

- Problem 10.1. Recursive Function Analysis

```cpp
// PRE: n >= 0
// POST: 2^n stars have been written to standard output
void g (const int n) {
    if (n == 0) {
        std::cout << "*";
        return;
    }
    g(n-1);
    g(n-1);
}
```

- The function g immediately terminates for n == 0. With each recursive call n is decremented, i.e., g is called with parameter < n, eventually reaching n = 0. This is true for both recursive calls of g.
- \( \text{Calls}_g(n) = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \)
Problem 10.2. Trains

i. \(<*-_*>\)\(<**>\) → invalid

ii. \(<*--------------------------*>\) → valid

iii. \(****-_\) → valid

iv. \(<*-------*>\)\(*----\) → invalid

→ \(*-_\)
Problem 10.2. Trains

// trains.cpp
// parse and validate train formations

#include <iostream>
#include <istream>
#include <sstream>
#include <string>

int main() {
    std::cout << "please enter a train formation:\n";
    // reading a string and wrapping it into stringstream prevents whitespace from occurring in a formation,
    // as can happen if reading directly from std::cin.
    std::string train_formation;
    std::cin >> train_formation;
    std::stringstream train_in(train_formation);
    if (train(train_in)) {
        std::cout << "valid\n";
    } else {
        std::cout << "invalid\n";
    }
    return 0;
}
Problem 10.2. Trains

// PRE: Valid stream is.
// POST: Leading whitespace characters are extracted from is, and the
// first non-whitespace character is returned (0 if there is none).
char lookahead (std::istream& is) {
    is >> std::ws; // skip whitespaces
    if (is.eof())
        return 0; // end of stream
    else
        return is.peek(); // next character in is
}

// PRE: Valid stream is.
// POST: Reads char from stream and returns true if that char matches
// the expected char, otherwise false is returned.
bool readchar(std::istream& is, char expected) {
    char actual;
    is >> actual;
    return actual == expected;
}
Problem 10.2. Trains

// train formations BNF:
// train = open | compositions .
// open = loco cars .
// loco = "*" | "*" loco .
// cars = "-" | "-"cars .
// compositions = composition { composition } .
// composition = "<" open loco ">".

bool train(std::istream& is) {
    // train = open | compositions .
    bool valid;
    if (lookahead(is) == '<')
        valid = compositions(is);
    else
        valid = open(is);
    // no more characters in string
    valid = valid && lookahead(is) == 0;
    return valid;
}
Problem 10.2. Trains

```cpp
bool open(std::istream& is) {
    // open = loco cars .
    return loco(is) && cars(is);
}

bool loco(std::istream& is) {
    // loco = "*" | "*" loco
    bool valid = readchar(is, '*');
    if (lookahead(is) == '*')
        valid = valid && loco(is);
    return valid;
}

bool cars(std::istream& is) {
    // cars = "-" | "-" cars
    bool valid = readchar(is, '-');
    if (lookahead(is) == '-')
        valid = valid && cars(is);
    return valid;
}
```
Problem 10.2. Trains

```cpp
bool compositions(std::istream& is) {
    // compositions = composition | { composition } .
    bool valid = composition(is);
    while(valid && lookahead(is) == '<') {
        valid = valid && composition(is);
    }
    return valid;
}

bool composition(std::istream& is) {
    // composition = "<" open loco ">".
    return readchar(is, '<') && open(is) && loco(is) && readchar(is, '>');
}
```
Problem 10.3. Money

```cpp
int main() {
    // the 13 denominations of CHF
    unsigned int chf[] =
        {100000, 20000, 10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5};

    // input
    std::cout << "in how many ways can I own x CHF-centimes for x =?\n";
    unsigned int x;
    std::cin >> x;

    // computation and output
    std::cout << partitions (x, chf, chf+13) << "\n";
    return 0;
}
```
Problem 10.3. Money

// PRE: [begin, end) is a valid nonempty range that describes
// a sequence of denominations d_1 > d_2 > ... > d_n > 0
// POST: return value is the number of ways to partition amount sing denominations from d_1, ..., d_n

unsigned int partitions (const unsigned int amount, const unsigned int* begin, const unsigned int* end) {

    if (amount == 0) return 1;
    unsigned int ways = 0;

    // ways = ways_1 + ... + ways_n, where ways_i is the number
    // of ways to partition amount using d_i as the largest denomination
    for (const unsigned int* d = begin; d != end; ++d)
        // ways_i = number of partitions of the form (d_i, X), with
        // (X) being a partition of amount-d_i using d_i,...,d_n
        if (amount >= *d)
            ways += partitions (amount-*d, d, end);

    return ways;
}
1 Structs

- 2.1 Data members

```c
struct rational {
    int n;
    int d; // INV: d != 0
};

int main () {
    rational r; // With C++11 initializer list: r = {1, 2}
    r.n = 1;
    r.d = 2;
    return 0;
}
```
1 Structs

- 2.1 Data members

```c
struct strange {
    int n;
    bool b;
    int a[3];
};

int main () {
    strange x;
    // With C++11 initializer list: x = {1, true, {1,2,3}};
    x.n = 1;
    x.b = true;
    x.a[0] = 1;
    x.a[1] = 2;
    x.a[2] = 3;
    strange y = x;
    return 0;
}
```
1 Structs

- Exercise 2) Struct for Point and Lines
2 Function Overloading

```c
int f (int a) { ... }

int f (int a, int b) { ... }

int f (float a) { ... }

int f (int b) { ... } // error

double f (int a) { ... } // error
```
2 Function Overloading

- Impact of Overloading

```cpp
double calculation (double value) { return value / 2.0; }
int calculation (int value) { return value / 2; }

int main () {
    int number;
    std::cin >> number;
    double result = calculation(number);
    std::cout << result;
    return 0;
}
```
3 Operator Overloading

- operators +=, -=, *=, /=

```c
struct rational {
    int n;
    int d; // INV: d != 0
};

rational& operator+=(rational& a, const rational b) {
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}

rational operator+(rational a, const rational b) {
    return a += b;
}
```
3 Operator Overloading

- ++r and r++

```cpp
// pre-comrement
rational& operator++ (rational& r) {
    rational s = {1,1};
    return r += s;
}

// post-comrement
rational operator++ (rational& r, int i) {
    rational s = {1,1};
    rational r_0 = r;
    r += s;
    return r_0;
}
```
3 Operator Overloading

- `<<`

```cpp
std::ostream& operator<<(std::ostream& o, rational r) {
    o << r.n << "/" << r.d;
    return o;
}
```
4 Tribool
5 Const references

```cpp
int a = 5;
int& b = a;
const int& c = a;
```

```cpp
c++; // runtime error, a cannot be changed through const reference c
b++; // a is now 6, a can still be changed through other non-const references
a++; // a is now 7, a can still change by itself
```

- Guarantee to not touch the variable
  ```cpp
  void print_result (const double& result) {...}
  ```

- Initialize with r-value
  ```cpp
  const int& a = 5; // compiler creates an int variable, puts 5 into it, creates a const reference pointing to it
  ```

- A const reference as an argument enables a function to do one of the two: const call by reference and call by value.
Übungsblatt 11

- Structs & Operators
- Problem 11.1 Finite Rings
  - Computations with modulo 7
  - Addition/Subtraction

- Problem 11.2a Complex numbers
  - Data type for complex numbers
  - Arithmetic operations

- Problem 11.2b Calculator for complex numbers
  - your own data type Complex as a drop-in replacement for double.