Principles of Programming Languages, 2012.07.23

Notes:
- Total available time: 2h.
- You may use any written material you need.
- You cannot use computers, phones or laptops during the exam.

Exercise 1 (8+8 pts)

a) Mushroom Inc. is a software consultant company specialized on functional programming. Its main product is based on "listoid" data structures, that are like lists and contain at least one element.
Please define a Haskell type class *Listoid* that contains all the types having the following operations:

- **cons**: given an element x and a listoid y, return a listoid having x as its first element and y as rest
- **unit**: given an element x, returns a listoid containing x
- **append**: given two listoids, returns the listoid concatenation of them
- **listoidfirst**: returns the first element of the listoid
- **listoidlast**: returns the last element of the listoid
- **listoidrest**: returns all the elements but the first of the input listoid - must return an error if called on a unit listoid

LL is one of such data structures: type *LL* represents lists optimized w.r.t. access to the first and the last element (i.e. accessing them has constant time complexity). Please define *LL* as an instance of *Eq*, *Listoid*, and *Show*.

b) Unfortunately for Mushroom Inc., one of their customers does not like the Haskell language – it prefers C++. In order to avoid losing the customer, Mushroom Inc. has to port LL to C++.
While the interface is more or less the same, the C++ version is slightly different from the Haskell one – it must be coded according to C++ idioms. For example, the *unit* operation can be replaced by a constructor.

You are required to build a first prototype of the C++ version of LL. The goal of your project is to test whether its interface matches customer requests, so you can select whichever implementation strategy you want – e.g. exploiting an already available data structure.

Exercise 2 (8 pts)

Consider the *proto-oo* system presented in class . Please implement a *chat* command for performing a "dialogue" between two objects, say *o1* and *o2*.
For example the command *(chat o1 o2 x m1 m2 ...)* must send the message *m1* with argument *x* to *o1*, obtaining a result. Such result is then used as argument of method *m2* of *o2*, and so on.
This means that, using a C++-like notation, it performs the following calls: *...o1.m3(o2.m2(o1.m1(x)))...*

Exercise 3 (8 pts)

Please implement the Prolog predicate *countpreds*, which has two inputs: an atom x, and a binary tree t. Such predicate must return the number of times that x is used as an internal node in t.
E.g.

if x = a, t = a(1,a(c(1,2),a(1,1))), *countpreds* returns 3;
if x = a, t = a(1,a(c(1,2),a)), *countpreds* returns 2.

Solutions

Ex 1 a)

```
class Listoid l where
    listoidcons   :: a -> l a -> l a
    listoidunit   :: a -> l a
    listoidappend :: l a -> l a -> l a
    listoidfirst  :: l a -> a
    listoidlast   :: l a -> a
    listoidrest   :: l a -> l a

newtype LL a = LL ([a], a) deriving Eq -- the second component is the last element


instance Listoid LL where
    x `listoidcons` LL (xs, y) = LL (x:xs, y)
    listoidunit x = LL ([], x)
    listoidappend (LL (x,x')) (LL (y,y')) = LL (x++[x']++y, y')
    listoidfirst (LL (x:xs, y)) = x
    listoidlast  (LL (x,y)) = y
    listoidrest  (LL (x:xs, y)) =
        if null xs then error "listoidrest on unit"
                   else LL (xs, y)


instance (Show a) => Show (LL a) where
    show (LL (x,y)) =
        (foldl (\x -> \y -> x ++ " " ++ y) "LL" (map show x)) ++ " " ++ show y
```

Ex 1 b) Here is an extended solution in two variants, one low level, and one using standard containers:

```
#include <iostream>
#include <iterator>
#include <list>
#include <vector>
#include <cstdlib>

namespace plp {

template <typename Ty>
class LL {
public:
  class Node {
  public:
    Node(const Ty &val) : val(val) { }

    Node(const Ty &val, Node *next) : val(val),
                                      next(next) { }

    Node(const Node &that); // Do not implement.
    const Node &operator=(const Node &that); // Do not implement.

  public:
    void setValue(const Ty &val) {
      this->val = val;
    }

    const Ty &getValue() const {
      return this->val;
    }
```

```
    void setNext(Node *node) {
      next = node;
    }

    Node *getNext() const {
      return next;
    }

  private:
    Ty val;
    Node *next;
  };

public:
  class const_iterator : public std::iterator<std::input_iterator_tag,
                                              const Ty> {
  public:
    const_iterator(const const_iterator &that) : cur(that.cur) { }

    const const_iterator &operator=(const const_iterator &that) {
      // Do not check for self-assignment -- faster and safe in this case.
      cur = that.cur;

      return *this;
    }

  private:
    const_iterator(const Node *cur = NULL) : cur(cur) { }

  public:
    bool operator==(const const_iterator &that) {
      return cur == that.cur;
    }

    bool operator!=(const const_iterator &that) {
      return !(*this == that);
    }

  public:
    const Ty &operator*() const {
      return cur->getValue();
    }

    const Ty *operator->() const {
      return &cur->getValue();
    }

    const_iterator &operator++() {
      cur = cur->getNext();

      return *this;
    }

    const_iterator operator++(int ign) {
      const_iterator cur = *this; ++(*this); return cur;
    }

  private:
    const Node *cur;

    friend class LL;
  };

public:
  const_iterator begin() const {
    return const_iterator(first);
```

```
  }

  const_iterator end() const {
    return const_iterator();
  }

public:
  const Ty &front() const {
    return first->getValue();
  }

  const Ty &back() const {
    return last->getValue();
  }

public:
  LL(const Ty &val) {
    Node *node = new Node(val);

    first = node;
    last = node;
  }

  LL(const LL &that); // Do not implement.
  const LL &operator=(const LL &that); // Do not implement.

  ~LL() {
    Node *cur = first;

    while(cur) {
      Node *next = cur->getNext();

      delete cur;

      cur = next;
    }
  }

public:
  void push(const Ty &val) {
    first = new Node(val, first);
  }

  Ty pop() {
    Node *cur = first;
    first = first->getNext();

    Ty val = cur->getValue();

    delete cur;

    return val;
  }

  template<typename IterTy>
  void insert(IterTy i, IterTy e) {
    Node *cur = last;

    for(; i != e; ++i) {
      Node *node = new Node(*i);

      cur->setNext(node);
      cur = node;
    }

    last = cur;
```

```
  }

public:
  void dump() const {
    std::cerr << *this;
  }

private:
  Node *first;
  Node *last;
};

template <typename Ty>
std::ostream &operator<<(std::ostream &os, const LL<Ty> &list) {
  typedef typename LL<Ty>::const_iterator iterator;

  os << "[ ";
  for(iterator i = list.begin(), e = list.end(); i != e; ++i)
    os << *i << " ";
  os << "]";

  return os;
}

template <typename Ty, typename Cnt = std::list<Ty> >
class IdiomLL {
public:
  typedef typename Cnt::iterator iterator;
  typedef typename Cnt::const_iterator const_iterator;

public:
  iterator begin() {
    return vals.begin();
  }

  iterator end() {
    return vals.end();
  }

  const_iterator begin() const {
    return vals.begin();
  }

  const_iterator end() const {
    return vals.end();
  }

public:
  const Ty &front() const {
    return *first;
  }

  const Ty &back() const {
    return *last;
  }

public:
  IdiomLL(const Ty &val) {
    vals.push_back(val);

    first = vals.begin();
    last = vals.end();

    --last;
  }
```

```
  IdiomLL(const IdiomLL &that); // Do not implement.
  const IdiomLL &operator=(const IdiomLL &that); // Do not implement.

public:
  void push(const Ty &val) {
    vals.push_front(val);

    first = vals.begin();
    last = vals.end();

    --last;
  }

  Ty pop() {
    Ty val = *first;

    vals.pop_front();

    first = vals.begin();
    last = vals.end();

    --last;

    return val;
  }

  template <typename IterTy>
  void insert(IterTy i, IterTy e) {
    for(; i != e; ++i)
      vals.push_back(*i);

    first = vals.begin();
    last = vals.end();

    --last;
  }

public:
  void dump() const {
    std::cerr << *this;
  }

private:
  Cnt vals;
  iterator first;
  iterator last;
};

template <typename Ty>
std::ostream &operator<<(std::ostream &os, const IdiomLL<Ty> &list) {
  typedef typename IdiomLL<Ty>::const_iterator iterator;

  os << "[ ";
  for(iterator i = list.begin(), e = list.end(); i != e; ++i)
    os << *i << " ";
  os << "]";

  return os;
}

} // End namespace plp.

using namespace plp;

template <typename Ty>
void demo(Ty &cnt);
```

```
int main(int argc, char *argv[]) {
  LL<unsigned> low(5);
  IdiomLL<unsigned> idiom(5);

  demo(low);
  std::cerr << std::endl;
  demo(idiom);

  return EXIT_SUCCESS;
}

template <typename Ty>
void demo(Ty &cnt) {
  std::cerr << "Start: " << cnt << std::endl;

  cnt.push(3);
  cnt.push(2);

  std::cerr << "Push: " << cnt << std::endl;

  std::vector<unsigned> decaPrimes;

  decaPrimes.push_back(11);
  decaPrimes.push_back(13);

  cnt.insert(decaPrimes.begin(), decaPrimes.end());

  std::cerr << "Insert: " << cnt << std::endl;

  cnt.pop();
  cnt.pop();

  std::cerr << "Pop: " << cnt << std::endl;

  std::cerr << "Front: " << cnt.front() << std::endl;
  std::cerr << "Back: " << cnt.back() << std::endl;
}
```

Ex 2

```
(define-syntax chat
  (syntax-rules ()
                ((_ ob1 ob2 start msg)
                 (-> ob1 msg start))

                ((_ ob1 ob2 start m1 m2 ...)
                 (let ((v (-> ob1 m1 start)))
                   (chat ob2 ob1 v m2 ...)))))
```

Ex 3

```
countpreds(Name, Tree, C) :- atomic(Tree), !, C is 0.
countpreds(Name, Tree, C) :- Tree =.. [X,L1,L2], X = Name, !,
        countpreds(Name, L1, C1),
        countpreds(Name, L2, C2),
        C is C1+C2+1.
countpreds(Name, Tree, C) :- Tree =.. [X,L1,L2], !,
        countpreds(Name, L1, C1),
        countpreds(Name, L2, C2),
        C is C1+C2.
```