CS 3721: Programming Languages Lab

Lab #12: Implementing subtype polymorphism in C++.

The following context-free grammar describes the syntax of a nested list which may contain symbols (with strings as contents) and numbers (with integer numbers as contents).

```
<element> ::= <sym> | <num> | ( <elements> )
<elements> ::= epsilon | <element>  <elements>
```

The following recursive ML datatype can be used to implement the data structure (i.e., the AST) for the above syntax.

```
datatype element = Sym of string  | Num of int | List of elements
and elements = Empty | Multi of element * elements;
```

To implement such a data structure in C++, we need to define the following class hierarchy representing subtype relations.

```
#include <stdio.h>
class Element
{
  public:
    virtual void print() = 0;
};
class Elements   // define all the base classes up front
{
  public:
    virtual void print()=0;
};
class Symbol : public Element
{
  private:
    const char* content;
  public:
    Symbol(const char* _content) : content(_content) {}
    const char* get_content() { return content; }
    virtual void print() { printf(" %s ",content); }
};
class Number : public Element
{
  private:
    int value;
  public:
    Number(int _value) : value(_value) {}
    int get_value() { return value; }
    virtual void print() { printf(" %d ", value); }
};
class List : public Element
{
  private:
    Elements* content;
  public:
    List(Elements* _content) : content(_content) {}
    Elements* get_content() { return content; }
```

```
    virtual void print() { printf("("); content->print(); printf(")"); }
};
class Empty : public Elements
{
  public:
    virtual void print() {}
};
class Multi : public Elements
{
  private:
    Element * head;
    Elements *tail;
  public:
    Multi(Element* _head, Elements* _tail) : head(_head), tail(_tail) {}
    Element* get_head() { return head; }
    Elements* get_tail() { return tail; }
    virtual void print() { head->print(); tail->print(); }
};
```

Testing the above C++ classes with the following driver

```
int main()
{
  Element * res = new List(new Multi(new Symbol("abc"),
                                     new Multi(new Number(3), new Empty())));
  res->print();
}
```

yields the result "(abc 3)".

1. Given the following context-free grammar.

   ```
   <graph> ::= digraph {  <edges> }
   <edges> ::= epsilon | <edge> <edges>
   <edge> ::= (<sym> -> <sym>)
   ```

   Here *epsilon* stands for the empty string, and ¡sym¿ stands for all kinds of non-empty strings. The following ML datatype defines the data structure (AST) for its syntax.

   ```
   datatype graph = Digraph of edges
     and edges = EmptyEdge | MultiEdge of edge * edges
     and edge = Edge of string * string;
   ```

   Use C++ classes to implement the data structure of the above context-free grammar. Testing your C++ classes with the following driver

   ```
   int main()
   {
     DiGraph * res = new DiGraph(new MultiEdge(new Edge("abc","def"),
                                               new MultiEdge(new Edge("abc","h"),
                                                             new EmptyEdge())));
     res->print();
   }
   ```

   which should yield the result " (abc -> def) (abc -> h) ".