Aufgabe 5: Reguläre Ausdrücke (C++)

Implementierungsaufgabe. Ziel ist die Vereinfachung regulärer Ausdrücke. Teile der Aufgabe sind schon vorgegeben. Siehe "Sourcefiles".

Reguläre Ausdrücke

In der theoretischen Informatik haben sie schon reguläre Ausdrücke kennengelernt. Mittels regulärer Ausdrücke lassen sich reguläre Sprache beschreiben, wobei eine Sprache aus einer Menge von Wörtern besteht. Jedes Wort besteht aus einer endlichen Anzahl von Zeichen. In unserem Fall stellen wir Zeichen als `char` dar. Deshalb sind Wörter nichts anderes als Strings.

Wir betrachten folgende syntaktischen Konstrukte:

- `eps` -- "Epsilon" der leere String
- `phi` -- "Phi" die leere Sprache
- `c` -- das Zeichen `c`
- `r1 + r2` -- Alternative zwischen `r1` und `r2`
    - Auch geschrieben als `r1 | r2`
- `r1 r2` -- Verkettung/Konkatenation von `r1` mit `r2`
- `r*` -- Kleenesche Hülle, entweder leerer String oder beliebige Verkettung von `r`

Die Sprache `L(r)` beschrieben durch den regulären Ausdruck `r` ist definiert wie folgt:

- `L(phi) = {}`
- `L(eps) = { eps }`
- `L(c) = { c }`
- `L(r1 + r2) = { w | w in L(r1) oder w in L(r2) }`
- `L(r1 r2) = { w1 w2 | w1 in L(r1) und w2 in L(r2) }`
- `L(r*) = { w | w = eps oder w=w1 ... wn jedes wi in  L(r)  }`

Im ersten Schritt bilden wir die Syntax von regulären Ausdrücken nach C++ ab. Dazu verwenden wir Vererbung.

```
class RE {};
class Phi : public RE {};
class Eps : public RE {};
class Ch : public RE {
 private:
  char c;
 public:
```

```
  Ch (char _c) { c = _c; }
};
class Alt : public RE {
 private:
  RE* r1;
  RE* r2;
 public:
  Alt (RE* _r1, RE* _r2) { r1 = _r1; r2 = _r2; }
};
class Conc : public RE {
 private:
  RE* r1;
  RE* r2;
 public:
  Conc (RE* _r1, RE* _r2) { r1 = _r1; r2 = _r2; }
};
class Star : public RE {
 private:
  RE* r;
 public:
  Star (RE* _r) { r = _r; }
};
```

Z.B. die abgeleitete Klasse `Alt` repräsentiert die Alternativen zwischen zwei regulären Ausdrücken. Die Alternativen werden als private Instanzvariablen `r1` und `r2` gespeichert. Beides sind Zeiger weil, wie wir noch sehen werden, wir virtuelle Methoden auf der Basisklassen und den Ableitungen definieren werden. Der `Alt` Konstruktor bekommt als Argument zwei reguläre Ausdrücke in C++ Repräsentation und initialisiert die Instanzvariablen.

Standard Methoden auf Regulären Ausdrücken

```
class RE {
 public:
  virtual REType ofType()=0;
  virtual string pretty()=0;
  virtual bool containsEps()=0;
};
```

`ofType` liefert einen enum Wert, der die Art des regulären Ausdrucks kennzeichnet. `pretty` ist eine "pretty print" Methode, die den regulären Ausdruck in einen String übersetzt. `containsEps` liefert `true` falls in der Sprache, die vom dem reguläre Ausdruck beschrieben wird, sich der leere String befindet.

Obige virtuelle Methoden sind schon für die abgeleiteten Klassen definiert. Zur Lösung der Aufgaben dürfen Sie weitere (virtuelle) Methoden zur Klasse `RE` hinzufügen.

Als Beispiel der Definition der virtuellen Methoden, betrachten wir die abgeleitete Klasse `Alt`.

```
enum REType {
  PhiType, EpsType, ChType,
  AltType, ConcType, StarType };

class Alt : public RE {
 private:
  RE* r1;
  RE* r2;
 public:
  Alt (RE* _r1, RE* _r2) { r1 = _r1; r2 = _r2; }
  REType ofType() { return AltType; }
  string pretty() {
    string s("(");
    s.append(r1->pretty());
    s.append("+");
    s.append(r2->pretty());
    s.append(")");
    return s;
  }
  bool containsEps() {
    return (r1->containsEps() || r2->containsEps());
  }
};
```

Die Syntax von regulären Ausdrücken ist fest vorgegeben. Sprich für die Klasse `RE` gibt es keine weiteren Ableitungen, ausser den oben beschriebenen. Deshalb benutzen wir einen enum, wobei die Tags des enums jede Ableitung der Klasse `RE` beschreiben. Z.B. `AltType` beschreibt `Alt`, `ConcType` beschreibt `Conc` usw. Deshalb liefert `ofType` für die Ableitung `Alt` den Wert `AltType`.

Im Falle von `pretty` wird ersichtlich, wieso wir virtuelle Methoden verwenden. `pretty` wird für jede Alternative aufgerufen. Zur Laufzeit wir dann basierend auf der konkreten Instanz der Alternativen, die geeignete `pretty` Definition ausgewählt.

Die resultierenden Strings werden dann mit "+" verbunden. Beachte: Mittels Klammern "(" und ")" wird eine eindeutige Darstellunge garantiert. Z.B.

```
  RE* r = new Conc (new Ch('c'), new Alt(new Ch('a'), new Ch('b')));
  cout << r->pretty() << endl;
```

liefert `(c(a+b))`. Ohne die Klammern erhalten wir `ca+b` was nicht eindeutig ist. Natürlich könnten wir eine Präferenz zwischen den Alternativ-, Verkettungs- und Staroperator definieren, um die Anzahl der Klammern zu reduzieren (freiwillige Aufgabe).

Im Falle einer Alternative, ist der leere String enthalten, falls eine der beiden Alternativen den leeren String enthält. Die Methode `containsEps` setzt diese Vorgabe literal um.

Aufgabe: Vereinfachung Regulärer Ausdrücke

Betrachte folgenden regulären Ausdruck.

```
eps ((a*)* (phi + b))
```

Obiger Ausdruck kann vereinfacht werden in die Form

```
(a*) b
```

Beide Ausdrücke sind äquivalent. Der letztere Ausdruck ist aber in einfacher und verständlicherer Form.

Ziel ist einen regulärern Ausdruck soweit wie möglich zu vereinfachen. Dazu verwenden wir Vereinfachungsregeln formuliert in der Form

```
linkeSeite ==> rechteSeite
```

1. `eps r ==> r`
2. `r1 r2 ==> phi` falls `L(r1)={}` oder `L(r2)={}`
3. `r* ==> eps` falls `L(r)={}`
4. `(r*)* ==> r*`
5. `r + r ==> r`
6. `r1 + r2 ==> r2` falls `L(r1)={}`
7. `r1 + r2 ==> r1` falls `L(r2)={}`

Für jede der obigen Regeln gilt `L(linkeSeite) = L(rechteSeite)`.

Angewandt auf unser Beispiel.

```
      eps ((a*)* (phi + b))
==>1  (a*)* (phi + b)
==>4  a* (phi + b)
==>6  a* b
```

Weiteres Vorgehen

Wir erweitern Klasse `RE` wie folgt.

```
class RE {
 public:
  ...
  virtual bool isPhi()=0;
  virtual RE* simp() { return this; }
};
```

Methode `isPhi` liefert `true` falls der Ausdruck die leere Sprache beschreibt und ist schon vordefiniert.

Methode `simp` führt die Vereinfachungsregeln aus. Per Default liefert `simp` den ürsprünglichen Ausdruck. Ihre Aufgabe ist es `simp` in den abgeleiteten Klassen geeignet zu erweitern.

Als Beispiel betrachte man `Alt` und die Umsetzung der Regeln 6 und 7.

```
RE* simp() {

  // First, simplify subparts
  r1 = r1->simp();
  r2 = r2->simp();

  // Then, check if any of the simplification rules are applicable

  // 6. `r1 + r2 ==> r2` falls `L(r1)={}`
  if(r1->isPhi()) return r2;
  // 7. `r1 + r2 ==> r1` falls `L(r2)={}`
  if(r2->isPhi()) return r1;

  return this;
}
```

Etwas schwierger sind Regeln 3 und 4 im Falle von `Star`.

```
RE* simp() {

  // Simplify subparts
  r = r->simp();

  // Then, check if any of the simplification rules are applicable

  // 3. `r* ==> eps` falls `L(r)={}`
  if(r->isPhi()) {
    return new Eps();
  }
  // 4. `(r*)* ==> r*`
  if(r->ofType() == StarType) {
    return this->r;
  }

  return this;
}
```

Beachte: In obiger Beispielimplementierung ignorieren wir das Problem der Speicherverwaltung. Z.B. im Falle von Regel (6) `r1 + r2 ==> r2` falls `L(r1)={}` sollte jemand de von `r1` belegten Speicherbereich freigeben. In dieser Aufgabe dürfen Sie das Aufräumen der von regulären Ausdrücken Speicherplätze ignorieren. Als freiwillige Zusatzaufgabe überlegen Sie sich ein geeignetes Verwaltungsschema welches garantiert, dass der belegte Speicherplatz freigeben wird. Die Herausforderung hierbei ist zu organisieren wer und wann den Speicherplatz

freigeben soll.

Ihre eigentliche Aufgabe ist die Implementierung der noch fehlenden obigen Vereinfachungsregeln. Für Regel (5) `r + r ==> r` benötigen Sie eine Methode zum Test auf Gleichheit zwischen regulären Ausdrücken.

Die einfachste Implementierung wendet die `pretty` Funktion an und vergleicht dann die beiden daraus resultierenden Strings.

```
bool equals(RE* r1, RE* r2) {
  return r1->pretty( ) == r2->pretty( );
}
```

Alternativ könnten wir rekursiv über den Strukturaufbau der regulären Ausdrücke laufen und die einzelnen Komponenten vergleichen.

```
bool equals(RE* r1, RE* r2) {
  bool b;

  if(r1->ofType( ) != r2->ofType( ))
     return false;

  switch(r1->ofType( )) {
    case PhiType: b = true;
                  break;
    case EpsType: b = true;
                  break;
    case ChType: {
                  Ch* c1 = (Ch*)r1;
                  Ch* c2 = (Ch*) r2;
                  b = c1->getChar( ) == c2->getChar( );
                  break;
    }
   case StarType: {
                   Star* r3 = (Star*) r1;
                   Star* r4 = (Star*) r2;
                   b = equals(r3->getRE( ),r4->getRE( ));
                   break;
   }
  case AltType: {
                Alt* r3 = (Alt*) r1;
                Alt* r4 = (Alt*) r2;
                b = equals(r3->getLeft( ),r4->getLeft( )) &&
            equals(r3->getRight( ), r4->getRight( ));
                break;
  }
  case ConcType: {
                Conc* r3 = (Conc*) r1;
                Conc* r4 = (Conc*) r2;
                b = equals(r3->getLeft( ),r4->getLeft( )) &&
            equals(r3->getRight( ), r4->getRight( ));
                break;
  }
  }// switch
  return b;
} // equals
```

Zusammengefasst. Erweiteren Sie `RE.h` mit den noch fehlenden Vereinfachungsregeln. Als eine mögliche Testroutine können Sie folgendes Gerüst verwenden.

```
// Main fuer Teilaufgabe 1
// Bitte weitere Testfaelle hinzufuegen.


#include "RE.h"
#include <iostream>

using namespace std;

int main() {

  // phi + c
  RE* r3 = new Alt (new Phi(), new Ch('c'));

  // c + phi
  RE* r4 = new Alt (new Ch('c'), new Phi());

  cout << r3->pretty() << endl;

  cout << r3->simp()->pretty() << endl;

  // c**
  RE* r5 = new Star(new Star (new Ch('c')));

  cout << r5->pretty() << endl;
  cout << r5->simp()->pretty() << endl;

  // phi*
  RE* r6 = new Star(new Phi());

  cout << r6->pretty() << endl;
  cout << r6->simp()->pretty() << endl;

  // (phi + c) + c**
  RE* r7 = new Alt(r3,r5);

  cout << r7->pretty() << endl;
  // exhaustively apply simplifications
  cout << simpFix(r7)->pretty() << endl;
}
```

Sourcefiles

```
// RE.h
// Reguläre Ausdrücke

#ifndef __RE__
#define __RE__

#include <string>
#include <sstream>

using namespace std;
```

```
// Vorwaertzreferenz
class RE;

// Prototypen von Hilfsfunktionen
bool equals(RE* r1, RE* r2);

enum REType {
  PhiType,
  EpsType,
  ChType,
  AltType,
  ConcType,
  StarType };


// Basisklasse
class RE {
 public:
  virtual REType ofType()=0;
  virtual string pretty()=0;
  virtual bool containsEps()=0;
  virtual bool isPhi()=0;
  virtual RE* simp() { return this; }
};

// Abgeleitete Klassen

class Phi : public RE {
 public:
  REType ofType() { return PhiType; }
  string pretty() { return "phi"; }
  bool containsEps() { return false; }
  bool isPhi() { return true; }
};

class Eps : public RE {
 public:
  REType ofType() { return EpsType; }
  string pretty() { return "eps"; }
  bool containsEps() { return true; }
  bool isPhi() { return false; }
};

class Ch : public RE {
 private:
  char c;
 public:
  Ch (char _c) { c = _c; }
  char getChar() { return c; }
  REType ofType() { return ChType; }
  string pretty() {
    stringstream ss;
    ss << c;
    return ss.str();
  }
  bool containsEps() { return false; }
  bool isPhi() { return false; }
};

class Alt : public RE {
 private:
  RE* r1;
  RE* r2;
```

```
 public:
  Alt (RE* _r1, RE* _r2) { r1 = _r1; r2 = _r2; }
  RE* getLeft() { return r1; }
  RE* getRight() { return r2; }
  REType ofType() { return AltType; }
  string pretty() {
    string s("(");
    s.append(r1->pretty());
    s.append("+");
    s.append(r2->pretty());
    s.append(")");
    return s;
  }
  bool containsEps() {
    return (r1->containsEps() || r2->containsEps());
  }
  bool isPhi() {
    return (r1->isPhi() && r2->isPhi());
  }
  RE* simp() {

    // First, simplify subparts
    r1 = r1->simp();
    r2 = r2->simp();

    // Then, check if any of the simplification rules are applicable

    // 6. `r1 + r2 ==> r2` falls `L(r1)={}`
    if(r1->isPhi()) return r2;
    // 7. `r1 + r2 ==> r1` falls `L(r2)={}`
    if(r2->isPhi()) return r1;

    return this;

    // N.B. We're a bit relaxed when it comes to garbage collection.
    // For example, in case of rule (6) we should clean up the
    // memory space occupied by r1 which we ignore here.
  }
};

class Conc : public RE {
 private:
  RE* r1;
  RE* r2;
 public:
  Conc (RE* _r1, RE* _r2) { r1 = _r1; r2 = _r2; }
  RE* getLeft() { return r1; }
  RE* getRight() { return r2; }
  REType ofType() { return ConcType; }
  string pretty() {
    string s("(");
    s.append(r1->pretty());
    s.append(r2->pretty());
    s.append(")");
    return s;
  }
  bool containsEps() {
    return (r1->containsEps() && r2->containsEps());
  }
  bool isPhi() {
    return (r1->isPhi() || r2->isPhi());
  }
};
```

```
class Star : public RE {
 private:
  RE* r;
 public:
  Star (RE* _r) { r = _r; }
  RE* getRE( ) { return r; }
  REType ofType( ) { return StarType; }
  string pretty( ) {
    string s;
    s.append(r->pretty( ));
    s.append("*");
    return s;
  }
  bool containsEps( ) {
    return true;
  }
  bool isPhi( ) {
    return false;
  }
  RE* simp( ) {

    // Simplify subparts
    r = r->simp( );

    // Then, check if any of the simplification rules are applicable

    // 3. `r* ==> eps` falls `L(r)={}`
    if(r->isPhi( )) {
      return new Eps( );
    }
    // 4. `(r*)* ==> r*`
    if(r->ofType( ) == StarType) {
      return this->r;
    }

    return this;
  }
};


// Structural comparison among regular expressions
bool equals(RE* r1, RE* r2) {
  bool b;

  if(r1->ofType( ) != r2->ofType( ))
     return false;

  switch(r1->ofType( )) {
    case PhiType: b = true;
                  break;
    case EpsType: b = true;
                  break;
    case ChType: {
                  Ch* c1 = (Ch*)r1;
                  Ch* c2 = (Ch*) r2;
                  b = c1->getChar( ) == c2->getChar( );
                  break;
    }
   case StarType: {
                   Star* r3 = (Star*) r1;
                   Star* r4 = (Star*) r2;
                   b = equals(r3->getRE( ),r4->getRE( ));
                   break;
   }
```

```
  case AltType: {
                Alt* r3 = (Alt*) r1;
                Alt* r4 = (Alt*) r2;
                b = equals(r3->getLeft(),r4->getLeft()) &&
            equals(r3->getRight(), r4->getRight());
                break;
  }
  case ConcType: {
                Conc* r3 = (Conc*) r1;
                Conc* r4 = (Conc*) r2;
                b = equals(r3->getLeft(),r4->getLeft()) &&
            equals(r3->getRight(), r4->getRight());
                break;
  }
  }// switch
  return b;
} // equals



// Repeated application of simp until we reach a fixpoint
RE* simpFix(RE* r1) {
  RE* r2 = r1->simp();

  if(equals(r1,r2))
       return r1;

  return simpFix(r2);
}

#endif // __RE__
```