Softwarelabor Aufgaben

Übersicht

- Bearbeiten Sie bitte folgende Aufgaben. Sie haben das Softwarelabor bestanden, wenn Sie alle Aufgaben erfolgreich bearbeitet haben.
- Sämtliche Aufgaben müssen im Softwarelabor persönlich vorgestellt werden. Abnahme durch Prof. Dr. Sulzmann.
- Abgabefristen siehe ILIAS
- Es folgt eine kurze Beschreibung. Weitere Informationen entnehmen Sie bitte dem Source Code der zu jeder Aufgabe bereitgestellt wird (siehe ilias).

Aufgabe 1: Bitmanipulationen (C)

Vertauschen von Bytes

Schreiben Sie eine Funktion die das Low Byte (Bits 0-7) und das High Byte (Bits 8-15) vertauscht. Z.B. aus der Zahl 0x4020 wird die Zahl 0x2040.

Verwenden Sie folgenden Funktionsprototypen.

```
short int switchLowHighByte(short int i);
```

Serialisierung/Deserialisierung von Datenstrukturen

Gegenben sind zwei enum Datentypen.

```
typedef enum {
  Stop = 0,
  Start = 1,
  Finish = 5,
  Fail = 255
```

```
} Status;


typedef enum {
  One = 1,
  Fifteen = 15,
  Last = 255
} Numbers;
```

Ihre Aufgabe ist es jeweils Werte der beiden enums in ein Datenpacket der Groesse 16Bit zu packen (serialisieren). Werte des enums Status sollen dabei in das Low Byte und Werte des enums Numbers sollen in das High Byte gepackt werden.

Schreiben Sie eine weitere Funktion, die Werte der enums Status und Number aus einem 16Bit Wert entpackt (deserialisiert). Wir nehmen an, dass die enum Werte mittels der serialize Funktion verpackt wurden.

Verwenden Sie folgende Funktionsprototypen.

```
void serialize(Status s, Numbers n, short int* data);

void deserialize(short int data, Status* s, Numbers* n);
```

Hinweise

Die Schreibweise

```
typedef enum {
  Stop = 0,
  Start = 1,
  Finish = 5,
  Fail = 255
} Status;
```

vereint die Deklaration eines enum und die Einführung einer Abkürzung. Ausführlicher

```
enum StatusEnum {
  Stop = 0,
  Start = 1,
  Finish = 5,
  Fail = 255
};
typedef enum StatusEnum Status;
```

Tips und Tricks

Bitoperationen

Sollten immer auf `unsigned` ausgeführt werden. Lokaler Typcast!

Hexadezimal Darstellung

C unterstützt die Darstellung und Ausgabe von Zahlen in Hexadezimal Darstellung.

```
short int zahl = 0x2040;
printf("%x \n", zahl);
```

Dies ist eine Hilfe beim Testen obiger Funktionen.

Rückgabe als Referenz

`serialize` und `deserialize` verwenden die Methode *Rückgabe als Referenz*.

Testen auf erwartetes Ergebnis

Ein *Testfall* besteht im allgemeinen aus

1. Setzen der Eingabe
2. Ausführung
3. Protokollierung der Ausgabe

Der Testfall ist erfolgreich, falls die Ausgabe mit dem erwarteten Ergebnis übereinstimmt.

Im unseren Fall, ist es recht einfach, das erwartete Ergebnis zu Beschreibung, falls z.B. `switchLowHighByte` zweimal ausgeführt wird.

```
enum TestEnum {
  OK,
  FAIL
};
typedef enum TestEnum Test;

Test testLowHigh(short int i) {
```

```
  Test t;
  if(i == switchLowHighByte(switchLowHighByte(i)))
    t = OK;
  else
    t = FAIL;

  return t;
}
```

Als Aufgabe für Sie bleibt die Auswahl einer Reihen von signifikanten Testeingaben.

Beachte: Obige Testeigenschaft `testLowHigh` ist notwendig aber nicht hinreichend für die Korrektheit von `switchLowHighByte`.

- Notwendig weil falls `switchLowHighByte` korrekt ist die Testeigenschaft sicherlich git.
- Nicht hinreichend weil z.B. eine fehlerhafte Implementierung in der immer die Eingabe unverändert zurückgeliefert wird, die Testeigenschaft erfüllt, nicht aber die Anforderungen an `switchLowHighByte`.

Test Set-up serialize/deserialize

```
Test testSD(Status s, Numbers n) {
 Test t;
 short int data;
 Status s2;
 Numbers n2;

 // Test execution
 serialize(s, n, &data);
 deserialize(data, &s2, &n2);

 if(s2 == s && n2 == n) {
    t = OK;
 }
 else {
    t = FAIL;
 }
 return t;
}
```

Aufgabe 2: Erkennen von Mustern (C)

In dieser Aufgabe betrachten wir Strings und Manipulationen auf Strings. In C, ein String ist repräsentiert als eine Sequenz von alphanumerischen Zeichen.

```
char s1[10];
char* s2;
char* s3[5];
char s4[5][6];
char** s5;
```

- s1 beschreibt einen String fester Größe.
- s2 ist ein Zeiger (Pointer) auf das erste Element eines Strings. Der vom String belegte Speicherbereich wird im Regelfall dynamisch angelegt
- s3 beschreibt ein Array von 5 Strings, wobei die einzelnen Strings durch Zeiger beschrieben sind.
- s4 beschreibt ein Array von 5 Strings,wobie die einzelnen Strings von fester Größe sind.
- s5 beschreibt einen Zeiger auf einen String

WICHTIG: Das Ende des Strings wir gekennzeichnet durch das Nullterminator Zeichen:

```
'\0'
```

Erkennen des längsten Suffix nach einem vorgegebenen Muster

Erste Teilaufgabe ist es eine Funktion zu schreiben, die den längsten suffix (Endung) liefert, in der keine zwei hintereinander folgenden Doppelpunkte (::) vorkommen. Als Eingabe betrachte man folgende Strings

```
"Ha::ll::o"
"47::11"
```

Längste Endungen für obige Beispiele sind

```
"o"
"11"
```

Verwenden Sie folgenden Funktionsprototypen.

```
char* extract(char* input);
```

In obiger Funktion wird das Ergebnis als return Wert geliefert. Das Ergebnis, ein Zeiger, verweist auf die Stelle im Eingabestring die den längsten suffix (Endung) liefert, in der keine zwei hintereinander folgenden Doppelpunkte (::) vorkommen.

Als Variante, verwenden Sie als Rückgabenwert einen Pointer auf einen Pointer. Wieso reicht ein Pointer nicht aus?

```
void extract2(char* input, char** output)
```

Freiwillige Zusatzaufgabe:

Verallgemeinern Sie Ihre Lösung: Das Muster zur Berechnung des längsten Suffix ist ein zusätzlicher Parameter.

```
char* extract(char* pattern, char* input);
```

Zählen von Wörtern

Ein Wort ist eine Sequenz von Zeichen in der das Leerzeichen (white-space) nicht vorkommt. Aufgabe ist es eine Funktion zu schreiben, die die Anzahl der Wörter in einem String zählt.

Verwenden Sie folgenden Funktionsprototypen.

```
int count(char* input);
```

Eine Lösung für diese Aufgaben wird im Softwarelabor besprochen. Verwenden Sie diese Lösung, um Ihre eigene Lösung zu testen. Details folgen im Labor.

Aufsammeln von Wörtern

Aufgabe ist es Wörter in einem String aufzusammeln.

Verwenden Sie folgenden Funktionsprototypen.

```
int breakIntoWords(char *line, int maxwords, char *words[])
```

Die aufgesammelten Wörter sollen in dem Array von Strings words gespeichert werden. Die Größe von words ist fest vorgegeben durch maxwords. Sprich die maximale Anzahl der aufzusammelnden Wörter ist durch maxwords begrenzt. Die Anzahl der aufgesammelten Wörter wird als return Wert zurück geliefert.

Als Speicherplatz für die aufgesammelten Wörter soll der Eingabestring verwendet werden.

Überlegen Sie sich folgende Alternative, bei der der Speicherplatz für die aufgesammelten Wörter dynamisch angelegt wird. Was sind die Vor-/Nachteile?

Tips und Tricks

String Initialisierung

```
char s1[] = "Hallo";
char s2[] = { 'H', 'a', 'l', 'l', 'o' };
```

Die Initialisierung von `s2` enthält einen typischen Fehler. Was ist falsch?

Zur Erinnerung. Jeder String muss mit Null terminiert werden wobei wir die Null durch das Zeichen `'\0'` beschreiben. Diese Zeichen fehlt im Falle von ‘s2. Richtig muss es also lauten

```
char s1[] = "Hallo";
char s2[] = { 'H', 'a', 'l', 'l', 'o', '\0' };
```

Im Falle der Notation `"Hallo"` wird automatisch die Null angehängt.

Zählen von Wörtern

Wir entwickeln eine Musterlösung für diese Teilaufgabe. Wichtig dabei ist der Nachweis der Korrektheit.

Um ein erstes Verständnis zu bekommen betrachten wir mögliche Eingaben und die erwarteten Ausgaben.

```
count("") ==> 0
count("Hallo") ==> 1
count("  Hallo") ==> 1
count("  Ha llo") ==> 2
```

Bevor man mit der Implementierung anfängt ist es elementar wichtig genau die Aufgabenstellung zu analysieren. *WAS* genau ist gefordert? *WAS* sind typische Eingaben und die erwarteten Ergebnisse?

Das *WIE* implementiere ich die Aufgabenstellung ist hier erst einmal zweitrangig.

Deshalb entwerfen wir als erstes einen Testrahmen, mit dessen Hilfe wir unsere Implementierung testen können. Von der "Java" Vorlesung sind Sie mit Unit-Tests vertraut. Solch ein Unit-Test Framework steht uns hier nicht zur Verfügung (gibt es natürlich auch für C/C++). Den notwendigen Testrahmen haben wir uns aber schnell selber gebaut.

Zuerst beschreiben wir die möglichen Testergebnisse.

```
typedef enum {
  OK,
  FAIL
} Test;
```

Ein Testfall besteht aus einem String und dem erwarteten Ergebnis. Wir schreiben eine kleine Hilfsfunktion, die einen Testfall ausführt und auf das erwartete Ergebnis testet.

```
Test testCount(char* input, int expected) {
  Test t;

  if(expected == count(input)) {
    t = OK;
  }
  else {
    t = FAIL;
  }
  return t;
}
```

Wir werden sicherlich eine Reihe von Testfällen benötigen, deshalb fassen wir einen Testfall in einer Struktur zusammen.

```
typedef struct {
  char* input;
  int expected;
} TestCase;
```

Folgende Hilfsfunktion erlaubt uns die automatische Ausführung einer ganzen Reihe von Testfällen.

```
void runTests(int no, TestCase test[]) {
  Test t;
  int i;
```

```
  for(i=0; i < no; i++) {
    printf("Test %d: ", i);
    t = testCount(test[i].input, test[i].expected);
    if(OK == t)
      printf("OK \n");
    if(FAIL == t)
      printf("FAIL \n");
  }
}
```

Eingabe `no` beschreibt die Anzahl der Testfälle. Die eigentlichen Testfälle werden als Array `test` bestehend aus Strukturen des Typs `TestCase` übergeben. Auf jeden Testfall wenden wir die Hilfsfunktion `testCount` an.

In unseren Fall protokollieren wir die Testergebnisse durch Aufgabe auf der Konsole. In Industrieprojekten werden die Testergebnisse normalerweise auf Platte geschrieben und mit der Versionsnummer der getesteten Software versehen.

Hier ist eine Auswahl signifikanter Testfälle.

```
int main() {
  const int testNo = 9;
  TestCase tests[9] = {
       {"", 0},
       {"Hallo", 1},
       {"  Hallo", 1},
       {"Hallo", 1},
       {"  Hallo  ", 1},

       {"Hal lo", 2},
       {" Hal lo", 2},
       {"Hal lo ", 2},
       {" Hal lo  ", 2}
     };

  runTests(testNo,tests);
}
```

Den Testrahmen haben wir also, wir haben aber immer noch keine einzige Zeile der Funktion `count` implementiert! Keine Angst kommt gleich. Die wichtige Nachricht an dieser Stelle ist: Ein Grossteil der Entwicklungszeit wird nicht auf das *WIE* verwendet, der meiste Entwicklungsaufwand beschäftigt sich mit *WAS*:

- ... soll die Software eigentlich leisen?

- ... ging schief, wo liegt der Fehler?
- ...

Nun endlich zur eigentlichen Implementierung. Betrachten wir noch einmal die Beispieleingaben und -ausgaben.

```
count("") ==> 0
count("Hallo") ==> 1
count("  Hallo") ==> 1
count("  Ha llo") ==> 2
```

Im Falle des leeren Strings geben wir `0` zurück. Ansonsten suchen wir den Anfang eines Wortes? WIE machen wir das?

Entweder (a) wir befinden uns schon am Anfang eines Wortes, oder (b) das aktuelle Zeichen ist ein Leerzeichen.

Im Fall (b) überspringen wir alle Leerzeichen. Aha, wir haben die Aufgabe zerlegt in eine Teilaufgabe. Hier ist eine Funktion, die die Teilaufgabe (a) umsetzt.

```
char* leerzeichen(char* input) {
  while(*input == ' ')
    input++;
  return input;
}
```

Wir "iterieren" durch den String solange bis wir auf ein Zeichen treffen, welches nicht eine Leerzeichen ist. Rückgabe ist ein Zeiger auf das getroffene Nicht-Leerzeichen.

Betrachten wir Fall (a). Wir befinden uns am Anfang eines Wortes. Ein Wort besteht aus einer Sequenz von Nicht-Leerzeichen. Also iterieren wir durch den String bis wir auf ein Leerzeichen treffen. Das stimmt fast! Zu beachten ist hier der Spezialfall, dass wir das Ende des Strings erreichen. Fall (a) lässt sich wiederum sehr einfach mit Hilfe einer Funktion beschreiben.

```
char* zeichen(char* input) {
  while(*input != '\0' && *input != ' ')
    input++;
  return input;
}
```

Wir haben jetzt die einzelnen Teilaufgaben. Damit lässt sich nun die `count` Aufgabe einfach lösen.

Zuerst in Pseudo-Code:

Solange das Ende nicht erreicht ist:

1. Überspringe alle Leerzeichen
2. Fall Zeichen gefunden
   (a) zähle Anzahl gefunder Wörter hoch
   (b) Gehe zu Wortende

Hier ist die (literale) Umsetzung.

```
int count(char* input) {
  int cnt = 0;

  while(*input != '\0') {
    input = leerzeichen(input);
    if(*input != '\0') {
      cnt++;
      input = zeichen(input);
    }
  }
  return cnt;
}
```

Als Anhang der komplette Programmcode

```
#include "stdio.h"

// Ueberspringe alle Leerzeichen
// Rueckgabe ist Zeiger auf das erste Nichtleerzeichen
char* leerzeichen(char* input) {
  while(*input == ' ')
    input++;
  return input;
}

// Scanne durch string solange bis wir auf ein
// Leerzeichen oder das Ende des Strings treffen.
// Effektiv ueberspringen wir ein Wort.
// Rueckgabe: Zeiger auf Ende oder Leerzeichen.
char* zeichen(char* input) {
  while(*input != '\0' && *input != ' ')
    input++;
  return input;
}
```

```
int count(char* input) {
  int cnt = 0;

  // Solange das Ende nicht erreicht ist:
  // 1. Ueberspringe alle Leerzeichen
  // 2. Falls Zeichen gefunden
  //     (a) setze Zaehler hoch
  //     (b) Gehe zu Wortende

  while(*input != '\0') {
    input = leerzeichen(input);
    if(*input != '\0') {
      cnt++;
      input = zeichen(input);
    }
  }
  return cnt;
}


typedef enum {
  OK,
  FAIL
} Test;

Test testCount(char* input, int expected) {
  Test t;

  if(expected == count(input)) {
    t = OK;
  }
  else {
    t = FAIL;
  }
  return t;
}


typedef struct {
  char* input;
  int expected;
} TestCase;


void runTests(int no, TestCase test[]) {
  Test t;
```

```
  int i;

  for(i=0; i < no; i++) {
    printf("Test %d: ", i);
    t = testCount(test[i].input, test[i].expected);
    if(OK == t)
      printf("OK \n");
    if(FAIL == t)
      printf("FAIL \n");
  }
}


int main() {
  const int testNo = 9;
  TestCase tests[9] = {
       {"", 0},
       {"Hallo", 1},
       {"  Hallo", 1},
       {"Hallo", 1},
       {"  Hallo  ", 1},

       {"Hal lo", 2},
       {" Hal lo", 2},
       {"Hal lo ", 2},
       {" Hal lo  ", 2}
     };

  runTests(testNo,tests);

}
```

Aufgabe 3: String Klasse mit überladenen Operatoren (C++)

Erstellen Sie eine String-Klasse ohne zu Hilfenahme von STL, die es Ihnen erlaubt, den folgenden Code zu schreiben:

```
int main() {
  String s1;
  String s2("Hello");
  String s3(s2);
  s1 += s2; s2 = s3;
  cout << s2 << endl;
```

```
  cout << s2[ 2 ] << endl;
};
```

Beachte

- Intern soll der String als ein Zeiger auf ein Zeichen dargestellt werden (wie in C).
- Bei der Konstruktion, Kopieren und bei der Zuweisung soll der String dupliziert werden. Sie benötigen daher einen Kopierkonstruktor, Zuweisungsoperator und einen Destruktor
- Folgende Operaten sollen überladen werden: += (Konkatenation), = (Zuweisung) und [] (Arrayzugriff)

Tips und Hinweise

Für den Arrayzugriff `string[index]` (sprich die überladene Methode) verwenden Sie bitte die folgende Signatur.

```
char& operator[](int index)
```

Wieso wird eine Referenz auf zurückgegeben? Beachte dass eine Arrayzugriff auch auf der linken Seite einer Zuweisung vorkommen kann.

Unten finden Sie ein mögliches Grundgerüst.

```
#include <iostream>
using namespace std;


class String {
 private:
  // 'String' is represented internally as a plain C-style string.
  int size;
  char* str;
 public:
  String() {
    size = 0;
    str = new char[1];
    str[0] = '\0';
  }
  String(char c) {
    size = 1;
    str = new char[2];
```

```
    str[0] = c;
    str[1] = '\0';
  }
  ~String() { delete[] str; }

  // make friend, so we can access private members
  friend ostream& operator<< (ostream &out, String &s);

};

ostream& operator<< (ostream &out, String &s) {
  for(int i=0; i<s.size; i++) {
    out << s.str[i];
  }

  return out;
}


int main() {
  String s;
  String s2('H');

  cout << s << endl;
  cout << s2 << endl;

}
```

Aufgabe 4: Reguläre Ausdrücke und Automaten (C++)

Diese Aufgabe besteht aus folgenden Teilen:

1. Vereinfachung regulärer Ausdrücke
2. Transformation Regulärer Ausdrücke in endliche Automaten
3. Ausführung des endlichen Automaten

Teile der Aufgabe sind schon vorgegeben. Siehe Files `RE.h` und `FSA.h`. Ihre Aufgabe ist es, die fehlenden Teile zu vervollständigen. Die Grundgerüste der einzelnen Teilaufgaben finden Sie im ilias.

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.

Teilaufgabe 1: 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;
}
```

Teilaufgabe 2: Transformation Regulärer Ausdrücke in Automaten

Die nächste Aufgabe ist die Transformation von regulären Ausdrücken in Automaten. Ihre Aufgabe ist die Implementierung eines Transformationsalgorithmus der einen regulären Ausdruck in einen Automaten umwandelt.

Endliche Automanten

Zuerst ein kurze Wiederhohlung der grundlegenden Konzepte endlicher Automaten.

Ziel ist die Überprüfung, ob eine Eingabewort Teil der vom Automaten akzeptierten Sprache ist. Dazu werden nacheinander die einzelnen Zeichen des Eingabe-

wortes betrachtet. Je nach aktuellem Zustand und aktuellem Zeichen, ändert sich der Zustand des Automaten (Beachte: Wir betrachten hier auch spontante Zustandsübergange, auch epsilon Transitionen genannt). Falls sämtliche Zeichen des Eingabewortes betrachtet, sprich konsumiert wurden, und der Automat befindet sich in einem Finalzustand, dann ist das Eingabewort Teil der vom Automaten akzeptierten Sprache.

Formal ist ein endlicher Automat beschrieben duch

1. Einer Menge Sigma aller Zeichen die vom Automaten behandelt werden, oft wird Sigma auch als das Alphabet des Automaten bezeichnet.
2. Einer Menge Q aller Automantenzustände.
3. Einen Initialzustand, auch Startzustand genannt, oft bezeichnet als q0 oder s0.
4. Einer Menge F von Finalzuständen, auch End- oder Stopzustände, genannt. Initialzustand und Finalzustände sind natürlich Elemente der Menge Q.
5. Einer Menge T von Transitionen, auch Übergänge genannt, von einem Zustand in den anderen. Ein Übergang (Transition) kann auf folgende zwei Arten beschrieben werden.
    1. Konsumierung eines Zeichens aus Sigma.
    2. Spontaner Übergang, auch "epsilon" Transition genannt.

Beispiel:

```
Sigma = {a,b,c}
Q = {1,2,3,4,5}
Initialzustand = 1
Finalzustände = {4,5}
Transitionen:
   1 -----> 2
   1 --a--> 3
   1 --a--> 5
   2 --a--> 2
   2 --b--> 2
   2 --a--> 3
   3 --c--> 4
   5 --b--> 4
```

Für Eingabewort "ab" betrachten wir die Ausführung des Automaten.

Wir starten im Zustand 1. Unser Automat ist nichtdeterministisch, deshalb kann der Automat zu jeden Zeitpunkt in einer Menge von "aktiven" Zuständen befinden.

Bei der Abarbeitung des Eingabewortes gibt es zwei prinzipielle Aktionen.

- Verfolge alle spontanen Übergange, d.h. wir bilden die Hülle (englisch closure) aller spontanten Übergänge.
- Konsumiere aktuelles Zeichen des Eingabeworts

```
1. current = { 1 }

Als erstes betrachten wir alle spontanen (epsilon) Übergange
die von 1 ausgehen.

Daraus folgt current = {1, 2}
wegen      1 -----> 2


Aktuelles Eingabezeichen ist a, die erste Position in "ab".

Daraus folgt current = {2,3,5} wegen
    1 --a--> 3
    1 --a--> 5
    2 --a--> 3
    2 --a--> 2

2. current = {2,3,5}

 Hülle der spontanen Übergänge ist hier gleich current

Aktuelles Eingabezeichen ist b, die zweite Position in "ab".

Daraus folgt current = {2,4} wegen
    2 --b--> 2
    5 --b--> 4

Da 4 ein Finalzustand ist, gehört "ab" zur der vom
dem Automaten akzeptierten Sprache.
```

Von regulären Ausdrücken zu endlichen Automaten

Endliche Automaten haben eine sehr einfache operationelle Beschreibung als eine Sequenz von Transitionsübergängen. In vielen Situationen sind jedoch reguläre Ausdrücke der geeignetere Formalismus. Beide Formalismen sind gleichwertig, d.h. ihre Ausdrücksstärke ist gleichwertig. Sprich jeder reguläre Ausdruck kann in einen endlichen Automaten umgewandelt werden und umgekehrt. Hier betrachten wir nur die Richtung von regulären Ausdrücken nach endlichen Automaten. Im folgenden betrachten wir einen speziellen Algorithmus der einen regulären Ausdruck in einen Automaten umwandelt (transformiert).

Thompson NFA Algorithmus Ein möglicher Algorithmus ist der Thompson NFA Algorithmus beschrieben hier:

http://en.wikipedia.org/wiki/Thompson's_construction_algorithm

Gegeben sein ein regulärer Ausdruck. Der Algorithmus von Ken Thompson zeigt wie man aus dem regulärern Ausdruck einen Automaten konstruiert. Der Automat ist nichtdeterministischen weil mehrere Folgezustände nichtdeterministisch erreicht werden können. Deshalb NFA, was auf Englisch heisst "non-deterministic finite automata".

Sie können natürlich sonst einen Algorithmus verwenden. Wichtig ist, dass der konstruierte Automat die gleiche Sprache akzeptiert, wie der reguläre Ausdruck. Zum Testen, ob Ihre Transformation diese Kriterium erfüllt wird Ihnen eine Testrahmen zur Verfügung gestellt (siehe unten).

C++ Umsetzung Folgende Klassen sind gegeben. Bauen Sie Ihre Lösung auf Grundlage dieser Klassen auf.

```
class Transition;
class NFA;
class FSA;
```

`Transition` beschreibt eine Transition innerhalb eines Automaten. `NFA` beschreibt eine nicht-deterministischen Automaten, der aus einer Menge von Transitionen, einem Startzustand und einer Menge von Finalzuständen besteht. `FSA` ermöglicht die Ausführung eines NFA und wird erst später relevant.

Zuerst betrachen wir die Klasse `Transition`.

```
class Transition {
 private:
  int from;
  char c;
  int to;
  bool epsilon;
 public:

  Transition(int _from, int _to) {
    from = _from; to = _to;
    epsilon = true;
  }
  Transition(int _from, char _c, int _to) {
    from = _from; c = _c; to = _to;
    epsilon = false;
  }
```

```
  bool isEpsilonTransition() { return epsilon; }
  int toState() { return to; }
  bool trigger(int from, char c) {
    return (!epsilon && from == this->from && c == this->c);
  }
  bool trigger(int from) {
    return (epsilon && from == this->from);
  }
};
```

Zustände werden als Integer Werte repräsentiert. Eine Transition kann zwei Formen haben.

```
from --- c ---> to

from ---------> to
```

Der Übergang von `from` nach `to` findet statt, falls die aktuelle Eingabe mit dem Zeichen `c` übereinstimmt, oder wir können spontan von `from` nach `to` übergehen. Letzer Fall ist eine sogeannte "epsilon" Transition. Die überladenen Konstruktoren von `Transition` erlauben die Konstruktion von Transitionen beider Arten.

Sind epsilon Transitionen notwendig? Nein. Jeder Automat mit epsilon Transitionen kann in einen Automaten ohne epsilon Transitionen. Jedoch sind epsilon Transitionen sehr hilfreich um nicht-deterministische Vorgänge einfach zu beschreiben. Siehe Aufgabe Übersetzung von regulären Ausdrücken nach Automaten.

Methode `isEpsilonTransition` unterscheidet zwischen beiden Arten. Überladene Methode `trigger` testet ob ein Übergang möglich ist. Methode `toState` liefert den Folgezustand.

Nun zu der `NFA` Klasse.

```
class NFA {
 private:
  vector<Transition> ts;
  int init;
  vector<int> final;

 public:
  NFA(vector<Transition> _ts, int _init, vector<int> _final) {
    ts = _ts;
    init = _init;
    final = _final;
```

```
  }
  NFA(vector<Transition> _ts, int _init, int _final) {
    ts = _ts;
    init = _init;
    final.push_back(_final);
  }
  vector<Transition> getTransitions() { return ts; }
  int getInitial() { return init; }
  vector<int> getFinals() { return final; }

  friend class FSA;
};
```

Ein `NFA` besteht aus einer Menge von Transitionen, einem Startzustand und einer Menge von Finalzuständen. Wir benutzen STL `vector` zur Speicherung von Transitionen und Finalzuständen. Der zweite Konstruktor erleichert die Erstellung eines NFAs mit genau einem Start und Endzustand.

Ihre Aufgabe ist die Implementierung einer Transformationsfunktion von regulären Automaten (RE) nach nicht-deterministischen Automaten (NFA).

Es gibt mindestens zwei mögliche Ansätze.

- Virtuelle Methode der Klasse `RE`
  - `RE.h` muss dann `FSA.h` importieren

```
class RE {
 public:
   virtual NFA transform()=0;
   // und restliche Methoden
};
```

- Eigenständige Funktion

```
NFA transform(RE* r);
```

Weitere Beispiele für beide Ansätze finden Sie unter Laboraufgaben (siehe Evaluator für arithmetische Ausdrücke).

Unten finden Sie ein Gerüst fuer den Ansatz “Eigenständige Funktion”. Die beiden Fälle `Phi` und `Alt` sind schon implementiert.

```
#include "FSA.h"
#include "RE.h"
```

```
int nameSupply;

void init() {
  nameSupply = 0;
}

int fresh() {
  return nameSupply++;
}


// Macht die eigentliche Arbeit
NFA transformWorker(RE *r);

// Schnittstelle fuer Benutzer
// Ruecksetzen des "name supplies" zur Generierung von eindeutigen Zustaenden
// Aufruf von transform2
NFA transform(RE *r) {
  init();
  return transformWorker(r);
}


// Wir liefern einen NFA zurueck mit genau einem Start und
// genau einem Stop(end)zustand.
NFA transformWorker(RE *r) {
  vector<Transition> ts;
  int start, stop;

  switch(r->ofType()) {

  case EpsType: // TODO
  case ChType: // TODO
  case StarType: // TODO
  case ConcType: // TODO

  // Phi: Akzeptiert kein Wort
  // NFA besteht aus einem Start und Stopzustand und *keiner* Transition
  case PhiType:
    start = fresh();
    stop = fresh();
    return NFA(ts, start, stop);

  case AltType: {
     Alt* r2 = (Alt*) r;
```

```
    // 1. Baue NFAs der linken und rechten Alternative
    NFA n1 = transformWorker(r2->getLeft());
    NFA n2 = transformWorker(r2->getRight());

    // 2. Generieren neuen Start-/Stopzustand.
    //    Sammle Start-/Stopzustaende von n1 und n2
    // N.B. Annahme: finals besteht aus genau einem Zustand
    start = fresh();
    stop = fresh();
    int n1_start = n1.getInitial();
    int n1_stop  = n1.getFinals()[0];
    int n2_start = n2.getInitial();
    int n2_stop  = n2.getFinals()[0];


    // 3. Sammle Transitionen auf von n1 und n2.
    //    Verbinde neuen Start-/Endzustand mit alten Start-/Endzustaenden.
    vector<Transition> t1 = n1.getTransitions();
    vector<Transition> t2 = n2.getTransitions();

    ts.insert(ts.end(),t1.begin(),t1.end());
    ts.insert(ts.end(),t2.begin(),t2.end());
    ts.push_back(Transition(start, n1_start));
    ts.push_back(Transition(start, n2_start));
    ts.push_back(Transition(n1_stop, stop));
    ts.push_back(Transition(n2_stop, stop));

    return NFA(ts,start,stop);
  }



  } // switch

} // transformWorker
```

Teilaufgabe 3: Ausführung NFA

Eine weitere Aufgabe ist die Ausführung des aus dem regulären Ausdrucks konstrukierten NFA. Der Rahmen zur Ausführung wird durch die Klasse `FSA` definiert.

```
class FSA : public NFA {
 private:
```

```
  vector<int> current;
  void closure();
 public:
  FSA(NFA fsa) : NFA(fsa.ts,fsa.init,fsa.final) {
    current.push_back(init);
    closure();
  }
  void reset();
  void step(char c);
  bool isFinal();
  bool run(string s);

};
```

Ein FSA nimmt einen NFA. Instanzvariable `current` beschreibt die Menge der "aktiven" Zustände. Beachte wir gehen von einem nicht-deterministischen Automaten aus. Im Falle eines deterministischen Automaten, würde die Menge aus genau einem Zustand bestehen.

Am Anfang wird `current` gleich dem Startzustand des NFA gesetzt. Übergänge finden nur statt (sprich Transitionen werden getriggert) bei Lesen eines Zeichens. Da der NFA "epsilon" Transitionen enthält, müssen wir alle von `current` aus erreichbaren Zustände bilden die via einer oder mehrerer epsilon Transitionenen erreichbar sind. Dies geschieht durch Aufruf der Methode `closure`.

`closure`

Ihre Aufgabe ist es Methode `closure` zu implementieren. Zuerst betrachten wir ein Beispiel mit folgenden drei Transitionen (Zustandsübergängen).

```
s1 -----> s2
s2 --a--> s3
s2 -----> s4
```

Von Zustand `s1` gibt es eine "epsilon" Transition nach `s2`. Von Zustand `s2` ist ein Übergang in den Zustand `s3` möglich, falls die aktuelle Eingabe `a` ist. Der letzte Zustandsübergang ist wieder ein "epsilon" Transition.

Annahme: `current = {s1}`

1. Schritt `current = {s1, s2}` wegen folgender "epsilon" Transition `s1 -----> s2`.

Aufgrund des neu hinzugefügten Zustandes `s2` können weitere Zustände via "epsilon" erreicht werden.

2. Schritt: `current = {s1, s2, s4}` wegen "epsilon" Transition `s2 -----> s4`.

Keine weiteren Zustände können via "epsilon" erreicht werden. Die `closure` von `{s1}` ist `{s1,s2,s4}`.

Beachte, Zustand `s3` kann nur erreicht werden falls die aktuelle Eingabe `a` ist. Deshalb ist `s3` nicht in der `closure` von `{s1}`.

Hinweise zur Implementierung von `closure`:

Für jeden Zustand `s1` in `current` und jede "epsilon" Transition `s2 -----> s3`, fügen wir `s3` zu `current` hinzu falls folgende Bedingungen gelten:

- 's1 == s2'
- `s3` ist nicht schon enthalten in `current`

Dieser Vorgang wird solange ausgeführt, bis keine Zustände mehr zu `current` hinzugefügt werden können.

Im Detail:

1. Iterieren Sie über die Menge der Transitionen.
2. Mit Hilfe der Methode `isEpsilonTransition` der Klasse `Transition` können Sie alle epsilon Transitionen herausfiltern.
3. Mit Hilfe der Methoden `trigger` und `toState` der Klasse `Transition` können Sie Testen, ob ein Zustand erreicht wird. Angenommen `t` ist eine Transition und `q` ein Zustand (Integer), dann liefert der Ausdruck `t.isEpsilonTransition() && t.trigger(q)` wahr, falls `t` eine epsilon Transition ist und Transition `t als Startpunkt den Zustandqhat. Via`t.toState(q)' können Sie den Endpunkt der (epsilon) Transition berechnen.
4. Um zu überprüfen, ob ein Element in einem Vektor ist benutzen Sie die `find` Method. Angenommen `v` ist vom Typ `vector<int>` und `q` ist ein Zustand, dann liefert der Ausdruck `find(v.begin(),v.end(),q) == v.end()` wahr, falls `q` nicht in dem Vektor `v` enthalten ist.

Weitere Methoden

Methoden `reset`, `isFinal`, `step` und `run` sind schon gegeben.

Ein `reset` bedeutet, wir brechen die Ausführung ab, und starten neu:

```
void FSA::reset() {
    current.clear();
    current.push_back(init);
    closure();
}
```

Methoden `isFinal` überprüft, ob wir einen Finalzustand erreicht haben:

```
bool FSA::isFinal() {
    for(int i=0; i < final.size(); i++) {
      if(find(current.begin(),current.end(),final[i]) != current.end())
    return true;
    }
    return false;
}
```

Methode `step` nimmt als Eingabe das aktuelle Zeichen `c`. Basierend auf der Menge der "aktiven" Zustände in `current`, werden alle Folgezustände berechnet die mit der Eingabe `c` erreicht werden können. Diese Menge wird gleich `current` gesetzt. Es folgt das bilden der transitiven Hülle der "epsilon" Transitionen mittels `closure`.

```
void FSA::step(char c) {
    vector<int> next;
    for(int i=0; i < ts.size(); i++) {
      for (int j=0; j < current.size(); j++) {
        if(ts[i].trigger(current[j],c))
      next.push_back(ts[i].toState());
      }
    }
    current = next;
    closure();
}
```

Methode `run` überprüft ob der Eingabestring von dem Automaten akzeptiert wird.

```
bool FSA::run(string s) {
  reset();
  for(int i=0; i < s.length(); i++) {
    step(s[i]);
  }
  return isFinal();
}
```

Testrahmen

Zum Testen der Teilaufgaben benutzen Sie bitte folgenden Testrahmen.

Annahme: Sie haben `FSA.h` mit einer Implementierung der `closure` Funktion erweitert und Teilaufgbe 1 as eingenständige Funktion folgender Form realisiert (falls als Methode, müssen Sie einfach den transform Aufruf geeignet umschreiben).

```
NFA transform(RE *r);
```

Testorakel

Zur Überprüfung, ob ihre Implemenierung korrekt ist benötigen wir eine unabhängige Implementierung eines Automaten.

Die untenstehende Implementierung beruht auf Brzozowski's Derivaten. Die Details sind unwichtig. Falls Sie's interessiert, weitere Informationen finden Sie http://lambda-the-ultimate.org/node/2293.

```
RE* deriv(RE* r, char l) {

  switch(r->ofType()) {
  case PhiType:
    return r;
  case EpsType:
    return new Phi();
  case ChType:
    if (((Ch*)r)->getChar() == l) {
      return new Eps();
    }
    else {
      return new Phi();
    }
  case StarType: {
    RE* r1 = ((Star*) r)->getRE();
    return new Conc(deriv(r1,l),r);
  }
  case AltType: {
    RE* r1 = ((Alt*) r)->getLeft();
    RE* r2 = ((Alt*) r)->getRight();
    return new Alt(deriv(r1,l), deriv(r2,l));
  }
  case ConcType: {
    RE* r1 = ((Conc*)r)->getLeft();
    RE* r2 = ((Conc*)r)->getRight();
```

```
    if(r1->containsEps()) {
      return new Alt(new Conc(deriv(r1,l),r2), deriv(r2,l));
    }
    else {
      return new Conc(deriv(r1,l),r2);
    }
  }

  }// switch

}

bool match(RE* r, string s) {
  for(int i=0; i < s.length(); i++) {
    r = deriv(r, s[i]);
  }
  return r->containsEps();
}
```

Testausführung

Ein Testfall besteht aus einem regulären Ausdruck und einem Eingabestring. Testausführung ist wie folgt.

1. Transformiere RE nach NFA
2. Baue FSA
3. Fuehre FSA fuer Eingabestring aus ("run")
4. Ueberpruefe ob Eingabestring in RE enthalten
5. Rueckgabe true falls Ergebnisse 3. und 4. uebereinstimmen. Sprich das Testorakel (Referenzimplementierung) und Ihre Implementierung liefern das gleiche Ergebnis. Ansonsten Rueckgabe false.

```
bool testExec(RE* r, string s) {
  NFA nfa = transform(r);
  FSA fsa(nfa);
  bool b1 = fsa.run(s);
  bool b2 = match(r,s);

  return b1 == b2;
}
```

main

Unten finden Sie ein paar einfache Testfälle. Sie sollten natürlich Ihre Implementierung gegen weitere Fälle testen.

```
int main() {

  // Testfaelle

  // (ab)*
  RE * r1 = new Star(new Conc(new Ch('a'), new Ch('b')));
  string s1 = "abab";

  cout << testExec(r1,s1) << endl;

  string s2 = "ababa";

  cout << testExec(r1,s2) << endl;
}
```

Anhang zu Aufgabe 4

Auch verfügbar via ilias.

FSA.h

```
// File: FSA.h
// Endliche Automaten

#ifndef __FSA__
#define __FSA__

#include <vector>
#include <string>
#include <algorithm>

using namespace std;

class Transition;
class NFA;
class FSA;
```

```
class Transition {
 private:
  int from;
  char c;
  int to;
  bool epsilon;
 public:

  Transition(int _from, int _to) {
    from = _from; to = _to;
    epsilon = true;
  }
  Transition(int _from, char _c, int _to) {
    from = _from; c = _c; to = _to;
    epsilon = false;
  }
  bool isEpsilonTransition() { return epsilon; }
  int toState() { return to; }
  bool trigger(int from, char c) {
    return (!epsilon && from == this->from && c == this->c);
  }
  bool trigger(int from) {
    return (epsilon && from == this->from);
  }
};

class NFA {
 private:
  vector<Transition> ts;
  int init;
  vector<int> final;

 public:
  NFA(vector<Transition> _ts, int _init, vector<int> _final) {
    ts = _ts;
    init = _init;
    final = _final;
  }
  NFA(vector<Transition> _ts, int _init, int _final) {
    ts = _ts;
    init = _init;
    final.push_back(_final);
  }
  vector<Transition> getTransitions() { return ts; }
  int getInitial() { return init; }
```

```
  vector<int> getFinals() { return final; }

  friend class FSA;
};

class FSA : public NFA {
 private:
  vector<int> current;
  void closure();
 public:
  FSA(NFA fsa) : NFA(fsa.ts,fsa.init,fsa.final) {
    current.push_back(init);
    closure();
  }
  void reset();
  void step(char c);
  bool isFinal();
  bool run(string s);

};

void FSA::reset() {
    current.clear();
    current.push_back(init);
    closure();
}

bool FSA::isFinal() {
    for(int i=0; i < final.size(); i++) {
      if(find(current.begin(),current.end(),final[i]) != current.end())
    return true;
    }
    return false;
}

void FSA::closure() {
 // Ihre Aufgabe
}

void FSA::step(char c) {
    vector<int> next;
    for(int i=0; i < ts.size(); i++) {
      for (int j=0; j < current.size(); j++) {
        if(ts[i].trigger(current[j],c))
      next.push_back(ts[i].toState());
      }
```

```
    }
    current = next;
    closure();
}

bool FSA::run(string s) {
  reset();
  for(int i=0; i < s.length(); i++) {
    step(s[i]);
  }
  return isFinal();
}


#endif // __FSA__
```

RE.h

```
// Reguläre Ausdrücke

#ifndef __RE__
#define __RE__

#include <string>
#include <sstream>

using namespace std;


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__
```

Transform.h

```
// Ein moeglicher Rahmen fuer Aufgabe 4, zweite Teilaufgabe,
// uebersetze regulaeren Ausdruck in einen NFA.
// Der Einfachheit in ein .h File gepackt.


#include <iostream>

using namespace std;

#include "FSA.h"
#include "RE.h"


int nameSupply;

void init() {
  nameSupply = 0;
}

int fresh() {
  return nameSupply++;
}


// Macht die eigentliche Arbeit
NFA transformWorker(RE *r);
```

```
// Schnittstelle fuer Benutzer
// Ruecksetzen des "name supplies" zur Generierung von eindeutigen Zustaenden
// Aufruf von transform2
NFA transform(RE *r) {
  init();
  return transformWorker(r);
}


// Wir liefern einen NFA zurueck mit genau einem Start und
// genau einem Stop(end)zustand.
NFA transformWorker(RE *r) {
  vector<Transition> ts;
  int start, stop;

  switch(r->ofType()) {

  case EpsType: // TODO
  case ChType: // TODO
  case StarType: // TODO
  case ConcType: // TODO

  // Phi: Akzeptiert kein Wort
  // NFA besteht aus einem Start und Stopzustand und *keiner* Transition
  case PhiType:
    start = fresh();
    stop = fresh();
    return NFA(ts, start, stop);

  case AltType: {
     Alt* r2 = (Alt*) r;

     // 1. Baue NFAs der linken und rechten Alternative
     NFA n1 = transformWorker(r2->getLeft());
     NFA n2 = transformWorker(r2->getRight());

     // 2. Generieren neuen Start-/Stopzustand.
     //    Sammle Start-/Stopzustaende von n1 und n2
     // N.B. Annahme: finals besteht aus genau einem Zustand
     start = fresh();
     stop = fresh();
     int n1_start = n1.getInitial();
     int n1_stop  = n1.getFinals()[0];
     int n2_start = n2.getInitial();
     int n2_stop  = n2.getFinals()[0];
```

```
    // 3. Sammle Transitionen auf von n1 und n2.
    //    Verbinde neuen Start-/Endzustand mit alten Start-/Endzustaenden.
    vector<Transition> t1 = n1.getTransitions();
    vector<Transition> t2 = n2.getTransitions();

    ts.insert(ts.end(),t1.begin(),t1.end());
    ts.insert(ts.end(),t2.begin(),t2.end());
    ts.push_back(Transition(start, n1_start));
    ts.push_back(Transition(start, n2_start));
    ts.push_back(Transition(n1_stop, stop));
    ts.push_back(Transition(n2_stop, stop));

    return NFA(ts,start,stop);
  }



  } // switch


} // transformWorker
```

TestOrakel.h

```
////////////////////////////////
// Testorakel
// der Einfachheit alles in ein .h File gepackt

#include "RE.h"

// Testorakel
RE* deriv(RE* r, char l) {

  switch(r->ofType()) {
  case PhiType:
    return r;
  case EpsType:
    return new Phi();
  case ChType:
    if (((Ch*)r)->getChar() == l) {
      return new Eps();
    }
    else {
```

```
      return new Phi();
    }
  case StarType: {
    RE* r1 = ((Star*) r)->getRE();
    return new Conc(deriv(r1,l),r);
  }
  case AltType: {
    RE* r1 = ((Alt*) r)->getLeft();
    RE* r2 = ((Alt*) r)->getRight();
    return new Alt(deriv(r1,l), deriv(r2,l));
  }
  case ConcType: {
    RE* r1 = ((Conc*)r)->getLeft();
    RE* r2 = ((Conc*)r)->getRight();
    if(r1->containsEps()) {
      return new Alt(new Conc(deriv(r1,l),r2), deriv(r2,l));
    }
    else {
      return new Conc(deriv(r1,l),r2);
    }
  }

  }// switch

}

bool match(RE* r, string s) {
  for(int i=0; i < s.length(); i++) {
    r = deriv(r, s[i]);
  }
  return r->containsEps();
}
```

testTeil1.cpp

```
// Testrahmen fuer 1. Teilaufgabe

#include "RE.h"
#include "FSA.h"

#include "TestOrakel.h"

#include <iostream>
#include <string>
```

```
using namespace std;


// Systematischer Test

// Ein Testfall besteht aus einem regulaeren Ausdruck (RE)
// und einem Eingabestring.
//
// Testorakel match ueberprueft ob string s enthalten in regex r.
//
// Simplifizierer korrekt fuer Testfall, falls Testorakel gleiches
// Ergebnis liefert fuer Original regex r und simplifizierten regex r->simp()

bool testSimp(RE* r, string s) {
  bool b = (match(r,s) == match(r->simp(),s));

  cout << "Test case: " << r->pretty() << "   " << s << "\n";
  cout << "Test result: " << b << endl;

  return b;
}

int main() {

  cout << "Ein paar Testfaelle. Ueberpruefung per Auge" << endl;


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

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

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

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

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

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

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

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

  cout << "Verwende testSimp" << endl;
```

```
  testSimp(r5, "ab");

  // TODO: mehr Tests


}
```

testTeil2und3.cpp

```
// Testrahmen fuer 2. und 3. Teilaufgabe

#include <iostream>

using namespace std;

#include "FSA.h"
#include "RE.h"

#include "Transform.h"
#include "TestOrakel.h"

#include <iostream>
#include <string>

using namespace std;


// Systematischer Test

// Ein Testfall besteht aus einem regulaeren Ausdruck (RE)
// und einem Eingabestring.
//
// Testorakel match ueberprueft ob string s enthalten in regex r.
//
// closure und Tranformater RE -> NFA korrekt, falls Testorakel gleiches
// Ergebnis liefert wie Ausfuehrung des resultierenden NFA.

bool testClosureTransform(RE* r, string s) {
  NFA nfa = transform(r);
  FSA fsa(nfa);
  bool b1 = fsa.run(s);
  bool b2 = match(r,s);
  bool b = b1 == b2;
```

```
  cout << "Test case: " << r->pretty() << "   " << s << "\n";
  cout << "Test result: " << b << endl;

  return b;
}

int main() {

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

    testClosureTransform(r3, "ab");

    // TODO: mehr Tests

}
```