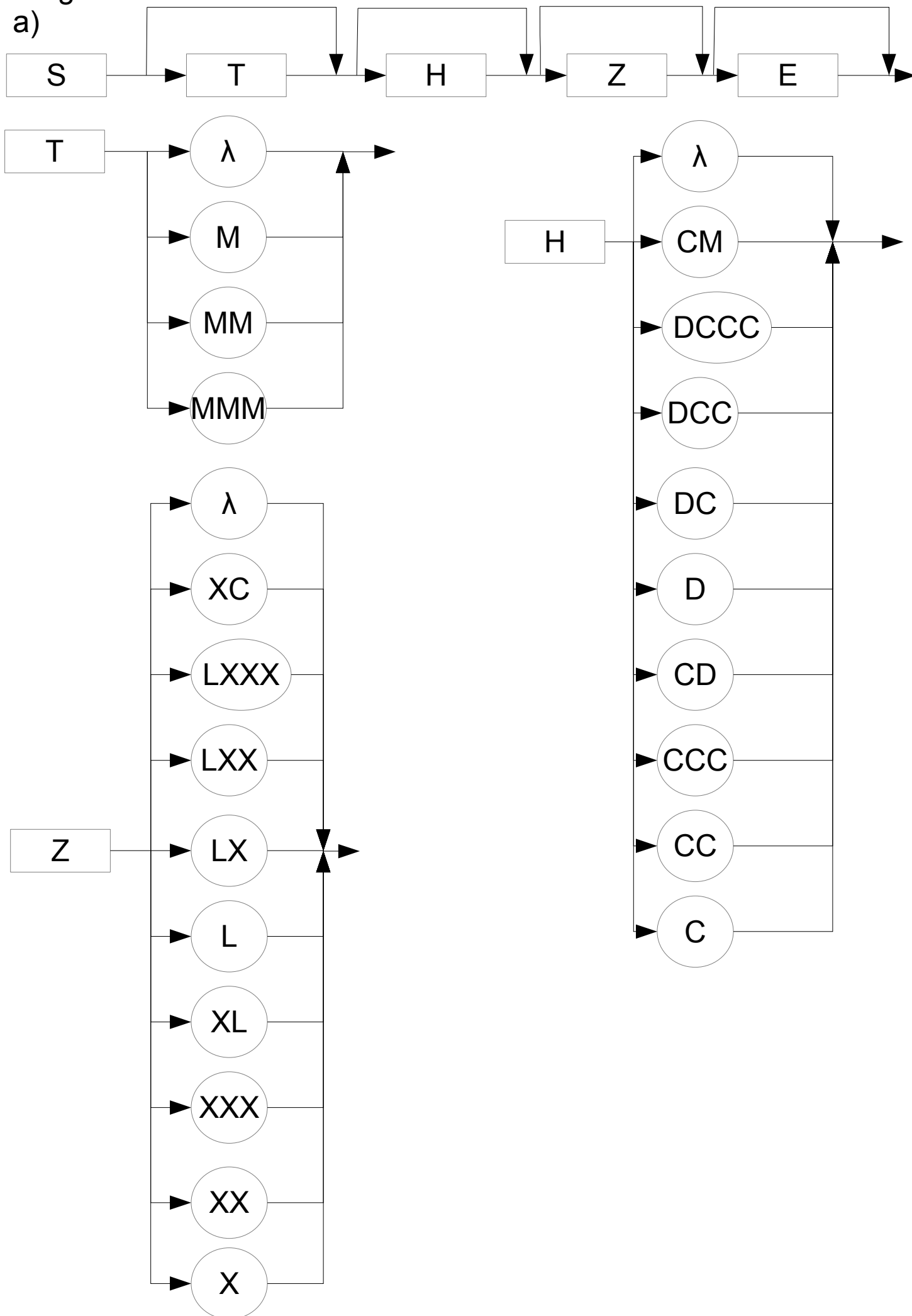
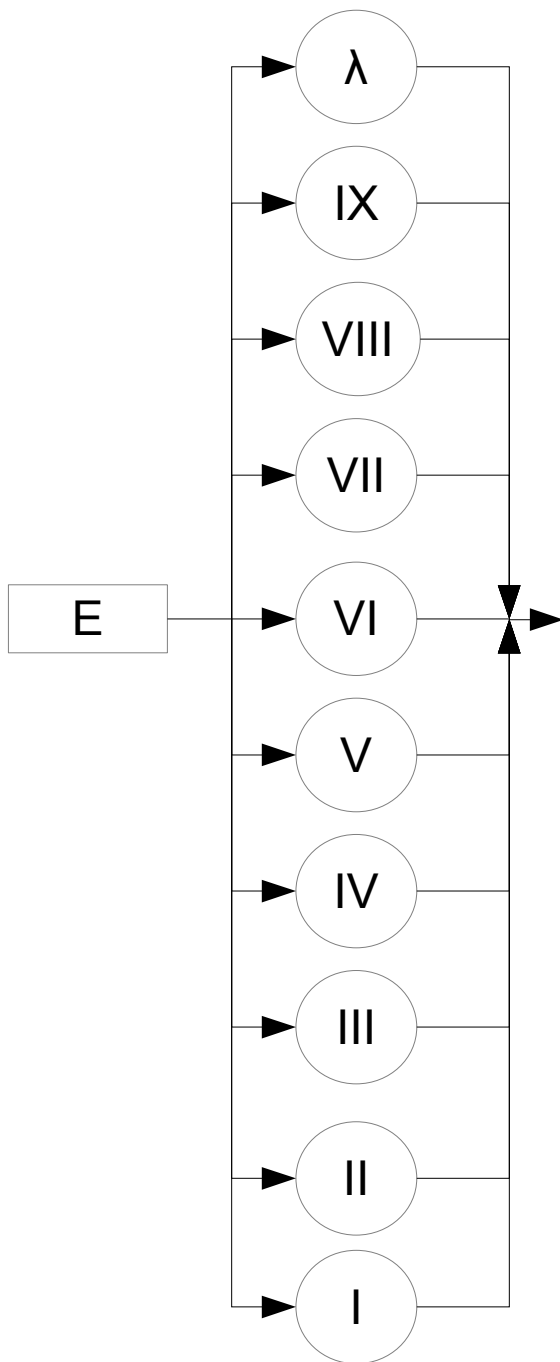


Aufgabe 4.1

a)





b) In diesem Fall kann man sich einfach das Maximum der Anzahl der Terminal-Symbole jedes Nicht-Terminals nehmen, also **10**. Bei jedem Durchlauf kann man dann ein anderes dieser 10 Knoten durchlaufen, wobei man bei den anderen Nicht-Terminals ebenfalls immer ein anderes nimmt.

c) Wie in a), nur dass jetzt noch die optionale Kante mitgenommen werden muss (die, die die Nicht-Terminals überspringen), da sonst für jedes Nicht-Terminal die Anzahl der Knoten und die Anzahl der Kanten gleich ist. Pro Nicht-Terminal hat man dann (Anzahl_der_Knoten + 1) Kanten und das Maximum ist **11**.

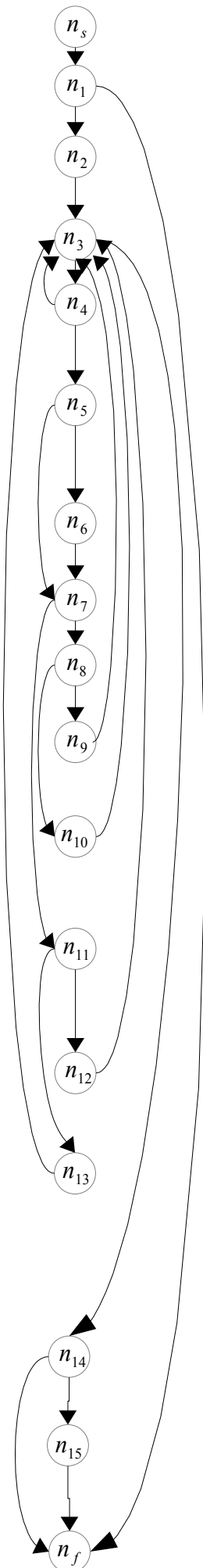
d) Alle Pfad-Möglichkeiten: $6 \cdot 11 \cdot 11 \cdot 11 = \mathbf{7.986}$

Aufgabe 4.2 a)

```

01  /** Method to analyse the elements of a given list of strings. */
02  public boolean analyseList(List<String> in_lstList, int[] out_aiResults) {
03      boolean bResult = false;
04      if (in_lstList != null && !in_lstList.isEmpty()) {
05          // initialize counters
06          int iNumberCounter = 0;
07          int iEvenNumberCounter = 0;
08          int iWordCounter = 0;
09          int iEvenWordCounter = 0;
10          // go through all elements of the given list
11          for (String strElement : in_lstList) {
12              // ignore 'null' and empty strings
13              if (strElement != null && !strElement.isEmpty()) {
14                  int iNumber = 0;
15                  boolean bNumberFound = false;
16                  // try to parse the current list element to an int value
17                  try {
18                      iNumber = Integer.parseInt(strElement);
19                      bNumberFound = true;
20                  } catch (NumberFormatException e) {
21                      // do nothing
22                  }
23                  // check if a number was found
24                  if (bNumberFound) {
25                      iNumberCounter++;
26                      // check if number is even or odd
27                      if (iNumber % 2 == 0) {
28                          iEvenNumberCounter++;
29                          System.out.println(iEvenNumberCounter +
30                              ". even number found: " + iNumber);
31                      } else {
32                          System.out.println(iNumberCounter +
33                              ". number found: " + iNumber);
34                      }
35                  } else { // no number found
36                      iWordCounter++;
37                      // check if the number of characters of the current
38                      // list element is even or odd
39                      if (strElement.length() % 2 == 0) {
40                          iEvenWordCounter++;
41                          System.out.println(iEvenWordCounter + ". word " +
42                              strElement + " with even number of characters found: " +
43                              strElement);
44                      } else {
45                          System.out.println(iWordCounter + ". word found: " +
46                              strElement);
47                      } // end if
48                  } // end if
49              } // end if
50          } // end for
51          // put the results into output array
52          if (out_aiResults != null && out_aiResults.length >= 4) {
53              out_aiResults[0] = iNumberCounter;    // numbers found
54              out_aiResults[1] = iEvenNumberCounter; // even numbers found
55              out_aiResults[2] = iWordCounter;      // words found
56              out_aiResults[3] = iEvenWordCounter;  // words with even number
57              bResult = true;                        // ...of characters found
58          } // end if
59      }
60      return bResult;
61  }

```



b)

Es werden die Knoten

n_s ,

$n_1, n_2, n_3, n_4, n_5, n_7, n_8$ (jeweils 4x)

n_9, n_{10} (jeweils 2x)

n_{14} und n_f

ausgeführt.

Dies sind insgesamt 12 von 17 Anweisungen und der Anweisungsüberdeckungsgrad ist demnach:

$$C_0 = \frac{12}{17}$$

Aufgabe 4.3:

a)

Testfall 1:

Leere Liste, leeres Array:

```
List<String> in_IstList = null;  
int[] out_aiResults = null;
```

Testfall 2:

Ein leerer String, eine gerade Zahl, eine ungerade Zahl, ein String gerader Länge und ein String ungerader Länge:

```
List<String> in_IstList = Arrays.asList("", "3", "42", "hallo", "welt");  
int[] out_aiResults = new int[4];
```

Testfall 3:

Wie Testfall 2, nur mit ungeeignetem Ergebnisarray.

```
List<String> in_IstList = Arrays.asList("", "3", "42", "hallo", "welt");  
int[] out_aiResults = new int[2];
```

b)

Primitive Zweige sind Zweige, die von der Ausführung anderer Zweige unabhängig sind. Ein primitiver Zweig ist der try-catch-Block. Wenn ein Integer geparsed werden konnte, dann wird automatisch auch der mit if(bNumberFound) beginnende Block ausgeführt.

Ein weiteres Beispiel wäre die for-Schleife. Diese stellt ebenfalls eine Verzweigung dar, da sie für eine leere Liste nichts tut und für eine nicht-leere Liste ihren Body für jedes Element ausführt. Die Bedingung if(in_IstList != null && !in_IstList.isEmpty()) gewährleistet aber bereits, dass die Liste nicht leer ist, der for-Zweig ist damit von dem primitiven Zweig abhängig.

c)

Da die Ausführung der primitiven Zweige aufgrund der oben beschriebenen Abhängigkeit die Ausführung aller nicht primitiven Zweige sicherstellt, ist es hinreichend nur die primitiven Zweige für die Berechnung des Überdeckungsmaßes zu nutzen.

d)

Zunächst stellt der Anweisungsüberdeckungstest eine echte Teilmenge des Zweigüberdeckungstests dar. Das bedeutet, wenn alle Zweige durchlaufen werden, dann werden auch alle Anweisungen ausgeführt. Der Zweigüberdeckungstest ist aber noch mächtiger, da er nicht nur die Ausführung der Anweisungen, sondern ebenfalls den Kontrollfluss testet und insbesondere durch Berücksichtigung der primitiven Zweige für die Berechnung der Abdeckung ein sinnvolles Maß bietet, da bei der Anweisungsüberdeckung die Abhängigkeiten der Anweisungen nicht berücksichtigt werden.

Aufgabe 4.4

Zunächst muss sichergestellt sein, dass das output-Array `out_aiResults` mindestens die Länge 4 hat, damit Anweisung 15 im Test enthalten ist.

Die Liste muss dann mindestens eines von jedem der folgenden Element-Typen enthalten:

- *Eine gerade Zahl
- *Eine ungerade Zahl
- *Ein Wort gerader Länge
- *Ein Wort ungerader Länge

Insgesamt gibt es also 4^{1000} Möglichkeiten die Liste aufzubauen, aus denen jeweils ein anderer Pfad resultiert. Nicht jeder Pfad ist nun ein vollständiger Pfad, diese Anzahl muss wieder abgezogen werden. Das sind all jene Listen-Möglichkeiten, die aus der Menge der 4 verschiedenen Element-Typen nur eine echte Teilmenge an Typen einbaut, also wenn mindestens ein Typ in der kompletten Liste fehlt. Praktisch wird dies jedoch kaum einen Unterschied machen, da diese Anzahl der unvollständigen Pfade im Gegensatz zur Gesamtanzahl viel kleiner ist:

$$\binom{4}{4} \cdot 4^{1000} - \binom{4}{3} \cdot 3^{1000} - \binom{4}{2} \cdot 2^{1000} - \binom{4}{1} \cdot 1^{1000} = 4^{1000} - 4 \cdot 3^{1000} - 6 \cdot 2^{1000} - 4 \approx 4^{1000}$$