Abgabetermin: Sa, 13.04.2024

☒ **Gr. 1,** S. Schöberl, MSc **Name** Elias Leonhardsberger **Aufwand in h** 9

☐ **Gr. 2,** DI (FH) G. Horn-Völlenkle, MSc

Punkte _____ **Tutor*in / Übungsleiter*in** ____ / ____

1. Wildcard Pattern Matching (4 + 5 + 5 Punkte)

Viele Programme, z. B. Texteditoren oder Kommandozeilen-Interpretierer diverser Betriebssysteme (engl. *shells, z. B. cmd* in Windows), verwenden eine spezielle Pattern-Matching-Variante, bei der es darum geht, festzustellen, ob eine Musterkette zu einer Zeichenkette passt. Denken Sie z. B. an den Windows-Befehl *del* *.* bzw. an das äquivalente UNIX-Kommando *rm* *. Hier muss festgestellt werden, ob die Musterkette (*.* bzw. *) zu einem Dateinamen im aktuellen Verzeichnis passt.

In Musterketten können so genannte Jokerzeichen (engl. *wildcards*) wie z. B. '?' oder '*' vorkommen. Dabei steht das Jokerzeichen '?' in der Musterkette für *ein* beliebiges Zeichen in der Zeichenkette und das Jokerzeichen '*' für eine *beliebige Anzahl* (null oder mehr) beliebiger Zeichen in der Zeichenkette. Diese Jokerzeichen können auch mehrfach in einer Musterkette vorkommen. Nehmen Sie an, dass sowohl die Muster- als auch die Zeichenkette durch das spezielle Endzeichen '$' abgeschlossen ist, wobei das Endzeichen nicht innerhalb der Ketten vorkommt.

Folgende Tabelle zeigt einige einfache *Beispiele*:

Musterkette *p*	**Zeichenkette *s***	***p* und *s* passen zusammen?**
ABC$	ABC$	ja
ABC$	AB$	nein
ABC$	ABCD$	nein
A?C$	AXC$	ja
*$	*beliebige auch leere Kette*	ja
A*C$	AC$	ja
A*C$	AXYZC$	ja

a) Erweitern/ändern Sie den *BruteForce*-Algorithmus so, dass er obige Aufgabenstellung bewältigt, jedoch als Jokerzeichen nur '?' (auch mehrfach) in der Musterkette vorkommen darf.

b) Die zusätzliche Behandlung des Jokerzeichens '*' ist mit den Standardalgorithmen leider nicht mehr so einfach möglich. Allerdings lässt sich das Problem relativ einfach mittels Rekursion lösen: (1) Definieren Sie zuerst ein rekursives Prädikat *Matching*(*p, s*), das *true* liefert, wenn *p* und *s* zusammenpassen, sonst *false*. Zerlegen Sie dabei sowohl *p* als auch *s* geschickt in zwei Teile: in das erste Zeichen und den Rest. (2) Implementieren Sie das Prädikat *Matching* in Form einer rekursiven Funktion und testen Sie diese ausführlich.

2. *m*-Ketten-Problem (4 + 6 Punkte)

a) *Definition*: Eine Zeichenkette ist eine *m*-Kette, wenn sie höchstens *m* unterschiedliche Zeichen enthält.

Beispiele: Die drei Zeichenketten *a*, *ab* und *abcbaac* sind 3-Ketten, die Zeichenkette *abcd* ist aber keine 3-Kette mehr, sondern eine 4-Kette. Eine Zeichenkette *s* der Länge *n* ist höchstens eine *n*-Kette, von Interesse ist aber das kleinste *m* für welches die Bedingung aus der Definition oben für *s* noch gilt. Entwickeln Sie daher eine möglichst effiziente Funktion

```
FUNCTION MFor(s: STRING): INTEGER;
```

zur Ermittlung des kleinsten *m* für die Zeichenkette *s*.

b) Gegeben sei eine nichtleere Zeichenkette *s* und eine ganze Zahl *m* mit $1 \leq m \leq Length(s)$. Entwickeln Sie eine möglichst effiziente Funktion

```
FUNCTION MaxMStringLen(s: STRING; m: INTEGER): INTEGER;
```

welche die Länge der längsten *m*-Kette, die als Teilkette in *s* enthalten ist, liefert.

Hinweise:

1. Geben Sie für alle Ihre Lösungen immer eine „Lösungsidee“ an.
2. Dokumentieren und kommentieren Sie Ihre Algorithmen ausführlich.
3. Bei Pascal-Programmen: Geben Sie immer auch Testfälle ab, an denen man sieht, dass Ihr Programm funktioniert, und dass es auch in Fehlersituationen entsprechend reagiert.

ADF2/PRO2 UE02

Elias Leonhardsberger

12. April 2024, Hagenberg

Inhaltsverzeichnis

1 Wildcard Pattern Matching

1.1 Lösungsidee

Da laut Angabe nur Strings mit gleicher Länge übereinstimmen können, kann man jedes Paar dessen Länge nicht übereinstimmt verwerfen und ab einem Unterschied den Vergleich, mit einem negativen Ergebnis, abbrechen.

Nach der Annahme, dass jeder Input mit einem $ Symbol abschließt, werden alle Inputs nicht damit enden verworfen.

Für Punkt b muss ein rekusiver Algorithmus verwendet werden. Es wird ein Zeichen überprüft und wenn dieses übereinstimmt werden die ersten Zeichen der Strings enfernt und die Funktion wieder aufgerufen. Wenn das erste Zeichen ein Symbol ist wird die Funktion dreimal aufgerufen, einmal mit dem verkürztem Pattern und Text, einmal mit nur verkürztem Pattern und einmal mit nur verkürztem Text. Dadurch werden die Fälle, ist ein Zeichen, ist mehrere Zeichen und ist kein Zeichen abgedeckt. Alle ungültigen Strings werden wieder verworfen.

1.2 Souce Code

1.2.1 SimpleWildCardMatcher.pas

```
1
2  UNIT SimpleWildCardMatcher;
3
4  INTERFACE
5  FUNCTION SimpleMatches(pattern, text: STRING): BOOLEAN;
6
7  IMPLEMENTATION
8  FUNCTION SimpleMatches(pattern, text: STRING): BOOLEAN;
9  VAR
10   i: INTEGER;
11   result: BOOLEAN;
12 BEGIN (* SimpleMatches *)
13   i := 1;
14   result := TRUE;
15
16   IF ((Length(pattern) = 0) OR (Length(text) = 0) OR
     ↪  (pattern[Length(pattern)] <> '$') OR (text[Length(text)] <> '$'))
     ↪  THEN
17     BEGIN (*IF*)
18       (* Invalid pattern or text *)
19       result := FALSE;
20     END; (*IF*)
21
22   IF ((result) AND (Length(pattern) = Length(text))) THEN
23     BEGIN (*IF*)
24       WHILE ((result) AND (i <= Length(pattern))) DO
25         BEGIN (*WHILE*)
```

```
            IF ((pattern[i] <> text[i]) AND (pattern[i] <> '?')) THEN
              BEGIN (*IF*)
                result := FALSE;
              END; (*IF*)
            i := i + 1;
          END; (*WHILE*)
      END (*IF*)
    ELSE
      BEGIN (*ELSE*)
        result := FALSE;
      END; (*ELSE*)

    SimpleMatches := result;
END; (* SimpleMatches *)

END.
```

1.2.2 RecursiveWildCardMatcher.pas

```

UNIT RecursiveWildCardMatcher;

INTERFACE
FUNCTION RecursiveMatches(pattern, text: STRING): BOOLEAN;

IMPLEMENTATION
FUNCTION RecursiveMatches(pattern, text: STRING): BOOLEAN;
VAR
  gotResult: BOOLEAN;
BEGIN (* RecursiveMatches *)
  gotResult := FALSE;

  IF ((Length(pattern) = 0) OR (Length(text) = 0) OR
  ↪  (pattern[Length(pattern)] <> '$') OR (text[Length(text)] <> '$'))
  ↪  THEN
    BEGIN (*IF*)
      (* Invalid pattern or text *)
      RecursiveMatches := FALSE;
      gotResult := TRUE;
    END; (*IF*)

  IF ((Length(pattern) = 1) AND (Length(text) = 1) AND (pattern[1] = '$')
  ↪  AND (text[1] = '$')) THEN
    BEGIN (*IF*)
      RecursiveMatches := TRUE;
      gotResult := TRUE;
    END; (*IF*)

```

```
27    IF (NOT gotResult) THEN
28      BEGIN (*IF*)
29        IF ((pattern[1] = '?') OR (text[1] = pattern[1])) THEN
30          BEGIN (*IF*)
31            RecursiveMatches := RecursiveMatches(Copy(pattern, 2,
                ↪  Length(pattern) - 1), Copy(text, 2, Length(text) - 1));
32          END (*IF*)
33        ELSE
34          IF (pattern[1] = '*') THEN
35            BEGIN (*IF*)
36              RecursiveMatches := RecursiveMatches(Copy(pattern, 2,
                  ↪  Length(pattern) - 1), Copy(text, 2, Length(text) - 1))
37                                  OR RecursiveMatches(pattern, Copy(text,
                                    ↪  2, Length(text) - 1))
38                                  OR RecursiveMatches(Copy(pattern, 2,
                                    ↪  Length(pattern) - 1), text);
39            END (*IF*)
40        ELSE
41          BEGIN (*ELSE*)
42            RecursiveMatches := FALSE;
43          END; (*ELSE*)
44      END; (*IF*)
45  END; (* RecursiveMatches *)
46
47  END.
```

1.2.3 TestWildCards.pas

```
1   PROGRAM TestWildCards;
2
3   USES
4   SimpleWildCardMatcher, RecursiveWildCardMatcher;
5
6   PROCEDURE TestSimpleMatches(p, s: STRING);
7   BEGIN (*TestSimpleMatches*)
8     WriteLn(p, ' ', s, ' ', SimpleMatches(p, s));
9   END; (*TestSimpleMatches*)
10
11  PROCEDURE TestRecursiveMatches(p, s: STRING);
12  BEGIN (*TestRecursiveMatches*)
13    WriteLn(p, ' ', s, ' ', RecursiveMatches(p, s));
14  END; (*TestRecursiveMatches*)
15
16  BEGIN (*TestWildCards*)
17    WriteLn('Simple Wildcard Matcher:');
18    TestSimpleMatches('ABC$', 'ABC$');
19    TestSimpleMatches('ABC$', 'AB$');
20    TestSimpleMatches('ABC$', 'ABCD$');
```

```
21    TestSimpleMatches('A?C$', 'AXC$');
22    TestSimpleMatches('A?C$', 'AXXC$');
23    TestSimpleMatches('A?C$', 'AC$');
24    TestSimpleMatches('A?C?E$', 'ABCDE$');
25    TestSimpleMatches('A?C?$', 'ABCD$');
26    TestSimpleMatches('ABC', 'ABC');
27    TestSimpleMatches('', '');
28    TestSimpleMatches('ABC', '');
29    TestSimpleMatches('', 'ABC');
30    TestSimpleMatches('$', '$');
31    TestSimpleMatches('', '$');
32    TestSimpleMatches('$', '');
33
34    WriteLn('Recursive Wildcard Matcher:');
35    TestRecursiveMatches('ABC$', 'ABC$');
36    TestRecursiveMatches('ABC$', 'AB$');
37    TestRecursiveMatches('ABC$', 'ABCD$');
38    TestRecursiveMatches('A?C$', 'AXC$');
39    TestRecursiveMatches('*$', '');
40    TestRecursiveMatches('*$', '$');
41    TestRecursiveMatches('*$', 'ABC$');
42    TestRecursiveMatches('*$', '*$');
43    TestRecursiveMatches('*$', '*');
44    TestRecursiveMatches('A*C$', 'AC$');
45    TestRecursiveMatches('A*C$', 'AXC$');
46    TestRecursiveMatches('A*C$', 'AXYZC$');
47    TestRecursiveMatches('A*C$', 'A$');
48    TestRecursiveMatches('A**C$', 'AC$');
49    TestRecursiveMatches('A**C$', 'AXC$');
50    TestRecursiveMatches('A**C$', 'AXYZC$');
51    TestRecursiveMatches('A**C$', 'A$');
52  END. (*TestWildCards*)
```

1.3 Testfälle

```
elias@EliasLaptop:~/Repos/SEbaBB2/pascal/PascalWorkspace/UE2/bin$ ./TestWildCards
Simple Wildcard Matcher:
ABC$ ABC$ TRUE
ABC$ AB$ FALSE
ABC$ ABCD$ FALSE
A?C$ AXC$ TRUE
A?C$ AXXC$ FALSE
A?C$ AC$ FALSE
A?C?E$ ABCDE$ TRUE
A?C?$ ABCD$ TRUE
ABC ABC FALSE
  FALSE
ABC  FALSE
 ABC FALSE
$ $ TRUE
 $ FALSE
$  FALSE
Recursive Wildcard Matcher:
ABC$ ABC$ TRUE
ABC$ AB$ FALSE
ABC$ ABCD$ FALSE
A?C$ AXC$ TRUE
*$  FALSE
*$ $ TRUE
*$ ABC$ TRUE
*$ *$ TRUE
*$ * FALSE
A*C$ AC$ TRUE
A*C$ AXC$ TRUE
A*C$ AXYZC$ TRUE
A*C$ A$ FALSE
A**C$ AC$ TRUE
A**C$ AXC$ TRUE
A**C$ AXYZC$ TRUE
A**C$ A$ FALSE
elias@EliasLaptop:~/Repos/SEbaBB2/pascal/PascalWorkspace/UE2/bin$
```

Abbildung 1: Diverse Tests zu Wildcard Pattern Matching

2 m-Ketten-Problem

2.1 Lösungsidee

Jedes Zeichen des Strings wird in ein Array mit dem Indextyp *CHAR* hinzugefügt und der Zähler erhöht. Der Datentyp im Array ist *BYTE*, obwohl für diese Aufgabe *BOOLEAN* reichen würde, aber für die nächste Aufgabe wird eine Zahl benötigt. *BYTE* wurde gewählt, da ein *STRING* nicht länger als 255 Zeichen sein kann, der Ausgabe Datentyp ist wie in der Angabe *INTEGER*. Wenn ein Zeichen davor schon in dem Array war wird der Zähler nicht mehr erhöht. Der Zähler wird am Ende ausgegeben. Der Ursprüngliche gedanke wäre eine Liste gewesen, da die Anzahl der Zeichen aber durch den Datentyp *CHAR* stark begrenz sind ist die Implementierung eines Arrays einfacher und schneller.

Laut der Angabe werden unterschiedliche Zeichen gezählt, also wird angenommen, dass A nicht das selbe wie a ist und, dass alle Zeichen erlaub sind.

Für den zweiten Teil der Angabe wird das Zählarray übernommen und um eine Remove Prozedur erweitert. Vom ersten Zeichen an wird überprüft wieviele Zeichen nach rechts die M-Ketten Bedingung erfüllen, ab dem nächsten Zeichen wird links von der Kette abgezogen bis die Bedingung wieder erfüllt ist. Dieses Verfahren wird wiederholt bis man am Ende vom String ankommt. Die maximale Länge wird am Ende zurückgegeben.

Das "nach rechts gehen"wird durch das Add gemacht und das "von links kommen"durch das "Remove".

2.2 Source Code

2.2.1 MChain.pas

```
1
2  UNIT MChain;
3
4  INTERFACE
5
6  FUNCTION MFor(s:STRING): INTEGER;
7  FUNCTION MaxMStringLen(s: STRING; m: INTEGER): INTEGER;
8
9  IMPLEMENTATION
10
11 TYPE
12   CountArray = ARRAY[CHAR] OF BYTE;
13
14 PROCEDURE InitCountArray(VAR ca: CountArray; VAR count: INTEGER);
15 VAR
16   i: CHAR;
17 BEGIN (*InitCountArray*)
18   count := 0;
19   FOR i := Low(CHAR) TO High(CHAR) DO
20     BEGIN (*FOR*)
21       ca[i] := 0;
22     END; (*FOR*)
23 END; (*InitCountArray*)
24
25 PROCEDURE CountArrayAdd(VAR ca: CountArray; c: CHAR; VAR count: INTEGER);
26 BEGIN (*CountArrayAdd*)
27   IF (ca[c] = 0) THEN
28     BEGIN (*IF*)
29       count := count + 1;
30     END; (*IF*)
31   ca[c] := ca[c] + 1;
32 END; (*CountArrayAdd*)
33
34 PROCEDURE CountArrayRemove(VAR ca: CountArray; c: CHAR; VAR ok: BOOLEAN;
   ↪  VAR count: INTEGER);
35 BEGIN (*CountArrayRemove*)
36   ok := FALSE;
37   IF (ca[c] <> 0) THEN
38     BEGIN (*IF*)
39       IF (ca[c] = 1) THEN
40         BEGIN (*IF*)
41           count := count - 1;
42         END; (*IF*)
43       ca[c] := ca[c] - 1;
```

```
44        ok := TRUE;
45      END; (*IF*)
46  END; (*CountArrayRemove*)
47
48  FUNCTION MFor(s: STRING): INTEGER;
49  VAR
50    i: INTEGER;
51    ca: CountArray;
52    count: INTEGER;
53  BEGIN (*MFor*)
54    InitCountArray(ca, count);
55    FOR i:=1 TO Length(s) DO
56      BEGIN (*FOR*)
57        CountArrayAdd(ca, s[i], count);
58      END; (*FOR*)
59    MFor := count;
60  END; (*MFor*)
61
62  FUNCTION MaxMStringLen(s: STRING; m: INTEGER): INTEGER;
63  VAR
64    i, j, count, maxLength: INTEGER;
65    ca: CountArray;
66    ok: BOOLEAN;
67  BEGIN (*MaxMStringLen*)
68    InitCountArray(ca, count);
69    i := 1;
70    j := 1;
71    maxLength := 0;
72    ok := TRUE;
73    WHILE ((ok) AND (j <= Length(s))) DO
74      BEGIN (*WHILE*)
75        IF (count <= m) THEN
76          BEGIN (*IF*)
77            CountArrayAdd(ca, s[j], count);
78            j := j + 1;
79            IF ((count <= m) AND (j - i > maxLength)) THEN
80              BEGIN (*IF*)
81                maxLength := j - i;
82              END; (*IF*)
83          END (*IF*)
84        ELSE
85          BEGIN (*ELSE*)
86            CountArrayRemove(ca, s[i], ok, count);
87            i := i + 1;
88          END; (*ELSE*)
89      END; (*WHILE*)
90
```

```
91    IF (ok) THEN
92      BEGIN (*IF*)
93        MaxMStringLen := maxLength;
94      END (*IF*)
95    ELSE
96      BEGIN (*ELSE*)
97        WriteLn('An error occured in MaxMStringLen');
98        MaxMStringLen := 0
99      END; (*ELSE*)
100 END; (*MaxMStringLen*)
101
102 END.
```

2.2.2 TestMChains.pas

```
1  PROGRAM TestMChains;
2
3  USES
4  MChain;
5
6  PROCEDURE TestMFor(s: STRING);
7  BEGIN (*TestMFor*)
8    WriteLn(s, ' ', MFor(s));
9  END; (*TestMFor*)
10
11 PROCEDURE TestMaxMStringLen(s: STRING; i :INTEGER);
12 BEGIN (*TestMaxMStringLen*)
13   WriteLn(s, ' ', i, ' ', MaxMStringLen(s, i));
14 END; (*TestMaxMStringLen*)
15
16 BEGIN (*TestMChains*)
17   WriteLn('MFor:');
18   TestMFor('a');
19   TestMFor('aa');
20   TestMFor('aaa');
21   TestMFor('ab');
22   TestMFor('abc');
23   TestMFor('abababababababa');
24   TestMFor('');
25   TestMFor('aabababababababac');
26   TestMFor('aA');
27   TestMFor('a1+#');
28
29   WriteLn();
30   WriteLn('MaxMStringLen:');
31   TestMaxMStringLen('a', 1);
32   TestMaxMStringLen('a', 2);
33   TestMaxMStringLen('a', 0);
```

```
34    TestMaxMStringLen('', 1);
35    TestMaxMStringLen('', 0);
36    TestMaxMStringLen('abababababa', 2);
37    TestMaxMStringLen('ababaababa', 1);
38    TestMaxMStringLen('abcabaabcaba', 2);
39    TestMaxMStringLen('abcabaabcaba', 3);
40    TestMaxMStringLen('abcabaabcaba', 3);
41    TestMaxMStringLen('ababacac', 2);
42    TestMaxMStringLen('babacaca', 2);
43    TestMaxMStringLen('+-+-++++', 2);
44    TestMaxMStringLen('-+-+++++', 2);
45  END. (*TestMChains*)
```

2.3 Testfälle

```
elias@EliasLaptop:~/Repos/SEbaBB2/pascal/PascalWorkspace/UE2/bin$ ./TestMChains
MFor:
a 1
aa 1
aaa 1
ab 2
abc 3
abababababaa 2
 0
aababababababababac 3
aA 2
a1+# 4

MaxMStringLen:
a 1 1
a 2 1
a 0 0
 1 0
 0 0
abababaa 2 9
ababaababa 1 2
abcabaabcaba 2 5
abcabaabcaba 3 12
abcabaabcaba 3 12
ababacac 2 5
babacaca 2 5
+-+-+*+* 2 5
-+-+*+*+ 2 5
elias@EliasLaptop:~/Repos/SEbaBB2/pascal/PascalWorkspace/UE2/bin$
```

Abbildung 2: Diverse Tests zum m-Ketten-Problem