ulm university universität ulm

2. Klausur zur Veranstaltung

Einführung in die Informatik II - Vertiefung

und **Allgemeine Informatik 2**

im Sommersemester 2022

Dr. Jens Kohlmeyer, Stefan Höppner

06.10.2022

Musterlösung

Aufgabe 1 - Verständnisfragen

3 + 6 + 3 = 12 Punkte

a) Erläutern Sie die Eigenschaften der Produktionsregeln, die für eine **Typ 3**-Grammatik gelten.

Lösung

- Nur ein Nicht-terminal auf linker Seite der Produktionsregel
- Es wird auf ein einzelnes Terminalsymbol abgebildet
- ODER Es wird auf ein einzelnes Terminalsymbol und Nichtterminalsymbol abgebildet, aber diese müssen immer in der selben Reihenfolge stehen (über alle Regeln hinweg)

b) Erläutern Sie **in Stichpunkten** einen rekursiven Algorithmus, der eine *depth-first* Suche auf einem Graphen durchführt.

Lösung

- Beginnend mit einem Startknoten wird jeweils entlang allen Kanten weiter gesucht
- nach folgen einer Kante wird der dahinterliegende Knoten als "visited" markiert
- bei jedem Schritt wird geprüft ob der aktuell besuchte Knoten "visited" ist
- wenn ja, wird die Suche für diesen Weg nicht weiter exploriert
- wenn nein, wird rekursiv weiter verfahren
- sind all vom aktuellen Knoten ausgehenden Kanten abgelaufen wird zum vorherigen Knoten zurückgegangen

c) Begründen Sie warum folgende Aussage nicht allgemein gültig ist:
"Ein Algorithmus mit einer Laufzeit von $O(n)$ ist schneller als ein Algorithmus mit einer Laufzeit von $O(n^2)$"
Verwenden Sie ein illustrierendes Beispiel zur Unterstützung Ihrer Erklärung.

Lösung

Die O-Notation verwirft konstante Faktoren **und** langsamer ansteigende Funktionsanteile.
Bsp.: $f_1 = n^2 + n$ vs. $f_2 = 300n$, f_2 ist für $n < 299$ langsamer.

Aufgabe 2 - Suchen

2.5 + 2.5 + 7 = 12 Punkte

a) Erklären Sie, warum bei einem Suchbaum nach dem Einfügen eines neuen Elements manchmal eine Umverteilung der Elemente vorgenommen werden sollte.

Lösung

Wenn dies nicht geschieht kann es vorkommen, dass die Tiefe des Baumes über verschiedene Pfade hinweg stark unterschiedlich ist. Dies kann im schlimmsten Fall alle Vorteile der Sortiereigenschaft des Baumes bei der Suche nach einem Element zerstören.

b) Erstellen Sie die delta1-Tabelle, welche für den Boyer-Moore-Algorithmus benötigt wird, für das Pattern *hanna*.

Lösung

Zeichen	Letztes Vorkommen
a	4
h	0
n	3

c) Suchen Sie mit dem Algorithmus von Boyer-Moore (BM) im unteren Text nach dem **ersten Vorkommen** des Patterns *hanna* aus Teilaufgabe **b)**. Füllen Sie für jedes Verschieben des Musters eine Zeile der nachstehenden Tabelle aus. Schreiben Sie in jede Zeile das gesamte Muster und **umkringeln** Sie alle Zeichen des Musters, die mit dem Text verglichen werden. Geben Sie für jeden Shift an, wie Sie diesen bestimmt haben. Notieren Sie dies **hinter** dem Muster auf dem Sie den Shift berechnet haben.
Hinweis: Sie benötigen nicht alle Zeilen der Tabelle.

Lösung

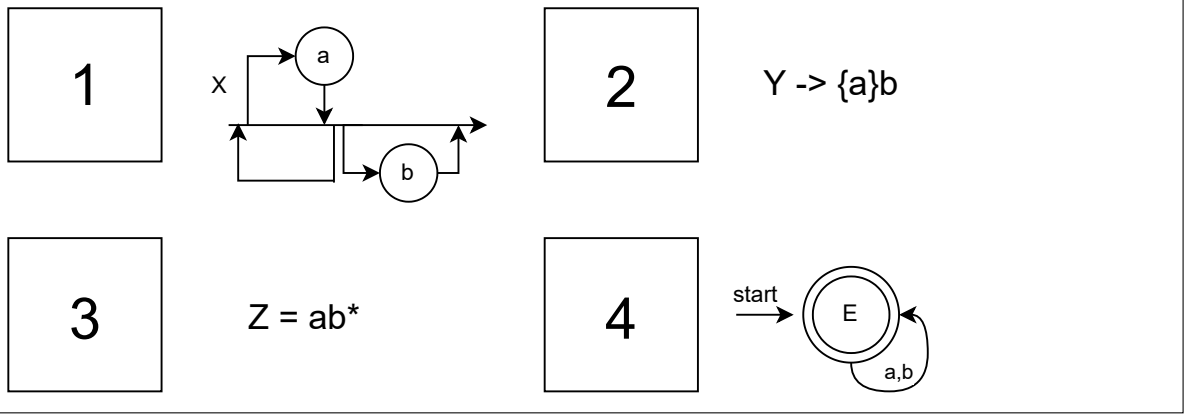
h	a	n	h	a	n	a	h	a	n	n		a
h	a	n	ⓝ	ⓐ	*Shift* $= 3 - 0 = 3$							
			h	a	n	n	a	ⓐ	*Shift* $= 4 - 0 = 4$			
							ⓗ	ⓐ	ⓝ	ⓝ		ⓐ

Aufgabe 3 - Formale Sprachen

4 + 3 + 5 = 12 Punkte

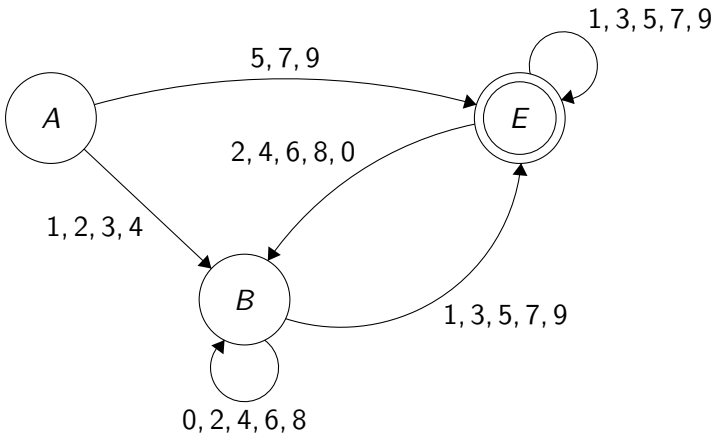
a) Im Folgenden sehen Sie einen regulären Ausdruck, einen endlichen Automaten, ein Syntaxdiagramm und eine EBNF. Markieren Sie, welche davon jeweils die gleiche Sprache erzeugen bzw. beschreiben, indem Sie in die Box eine Zahl eintragen. *Wären Sie zum Beispiel der Meinung, dass alle unterschiedliche Sprachen beschreiben bzw. erzeugen, dann tragen Sie in jede Box eine andere Zahl ein.*

Lösung



b) Geben Sie einen **deterministischen** endlichen Automaten an, der alle ungeraden natürlichen Zahlen größer 4 ohne führende Nullen akzeptiert.

Lösung



c) Die Programmiersprache LISP zeichnet sich durch eine sehr simple Syntax aus, die sich leicht in eine Grammatik und, daraus resultierend, leicht parsen lässt.
In dieser Aufgabe betrachten wir ein Subset valider LISP Ausdrücke, hier genannt LISP^-:
Jeder Ausdruck in LISP^- beginnt mit einer öffnenden und endet mit einer schließenden Klammer. Ein LISP^- Ausdruck ist eine Operation mit 2 bzw. 3 Argumenten in Präfixnotation (d.h der Name der Operation kommt vor den Operanden). Operationen bekommen immer entweder eine ganze Zahl ≥ 0 oder weitere LISP^- Ausdrücke als Operanden übergeben. Die erlaubten 2-Argument Operationen in LISP^- sind: $+$, -, * und $/$. Die einzige erlaubte 3-Argument Operation in LISP^- ist: *if*.
Definieren Sie eine Grammatik die alle gültigen LISP^- Ausdrücke akzeptiert.
Beispiel: Folgender Ausdruck ist ein valider LISP^- Ausdruck: $(* \ (+ \ 1 \ 2) \ (\mathit{IF} \ 0 \ (+ \ 1 \ 2) \ (- \ 900 \ 10)))$

Lösung

V = {S, OP, 2OP, C, 3OP, EXP, NBR}
Σ = {0,1,2,3,4,5,6,7,8,9,+,-,*,/,if,(,)}
S = {S}
P = {S → (OP)
OP → 2OP | 3OP
2OP → C EXP EXP
C → + | - | * | /
3OP → if EXP EXP EXP
EXP → S | NBR
NBR → 0 | ... | 9 | 1NRB | ... | 9NBR
}

Aufgabe 4 - Java - Datenstrukturen **4 + 3 + 5 = 12 Punkte**

Betrachten Sie folgendes Interface und die Beschreibung:

```
1 public interface BetterSet<T> {
2     public BetterSet<T> add(T elem);
3     public BetterSet<T> remove(T elem);
4     public BetterSet<T> union(BetterSet<T> set);
5 }
```

`BetterSet` definiert ein `Set` das Methodchaning unterstützt. Wie bei einem normalen `Set` werden Elemente nur hinzugefügt wenn sie noch nicht im Set existieren.

a) Definieren Sie eine Klasse `MySet<T>` welche das Interface `BetterSet<T>` implementiert. Implementieren Sie hierfür alle notwendigen Methoden **mit der Ausnahme von** `intersect`. Wählen Sie als interne Datenstruktur eine `LinkedList<T>`.

Lösung

```
1  public class MySet<T> implements BetterSet<T> {
2      private LinkedList<T> data = new LinkedList<>();
3
4      public MySet<T> add(T elem) {
5          if (!data.contains(elem)) {
6              data.add(elem);
7          }
8          return this;
9      }
10
11     public MySet<T> remove(T elem) {
12         data.remove(elem);
13         return this;
14     }
15 }
```

b) Erklären Sie einen Vorteil und einen Nachteil den `BetterSet` gegenüber eines "normalen" Sets hat. Demonstrieren Sie beides an je einem Beispiel.

Lösung

z.B.:
Vorteil: Kann mehrere Elemente nacheinander hinzufügen `myBetterSet.add(1).add(2)`
Nachteil: Man weiß nicht ob Operationen eine Änderung am Set vorgenommen haben `set.add(1)` → `true|false` je nachdem.

c) Gehen Sie davon aus, dass eine Implementierung von `BetterSetk<T>` als interne Datenstruktur ein Array verwendet. Beschreiben Sie, wie, auf Basis dieser Vorgabe, die Methode `union` zu implementieren wäre. Die Methode vereinigt die Inhalte der zwei involvierten Mengen in der aufgerufenen Menge und verwirft dabei doppelte Einträge.
Beispiel für die Darstellung:

i. Lese erstes Element aus interner Datenstruktur aus
ii. Prüfe ob es `null` ist
iii. ...

Lösung

i. Lese Länge der internen Datenstruktur
ii. Lese Länge der internen Datenstruktur des Übergabeparameters
iii. Erzeuge neues Array mit Länge von Summe der beiden
iv. Schleife über interne Datenstruktur und füge alle Elemente dem neuen Array hinzu mit pos intern = pos neu
v. Schleife über Datenstruktur des Parameters
vi. Schleife über neues Array
vii. prüfe für jedes Element in neuem Array ab, ob das aktuelle Element im Array vorkommt
viii. wenn ja continue
ix. wenn nein füge es an erster freier Stelle ein (länge internes Array + pos parameter z.B)
x. internes Array durch neues Array ersetzen
xi. return this

Aufgabe 5 - Java - Bekannte

4 + 2 + 6 = 12 Punkte

Betrachten Sie folgend Klassen, welche Bekanntenrelationen darstellt:

```
1 public class Person {
2     String name;
3     Set<Person> knows;
4 }
```

a) Implementieren Sie die Methode `mingle(Person[] persons)` der Klasse `Person`, welche alle Bekannten der übergebenen Personen `persons` zu den Bekannten der Person hinzufügt. Dies soll auch umgekehrt stattfinden.

Lösung

```
1 public void mingle(Person[] persons) {
2     for (Person p: persons) {
3         p.knows.addAll(this.knows);
4     }
5     for (Person p: persons) {
6         this.knows.addAll(p.knows);
7     }
8 }
```

b) Nennen und erläutern Sie **2** Gründe, warum für das Attribut `knows` ein `Set` geeigneter ist als ein Array.

Lösung Arrays haben eine Limitierte Länge. Da nicht initial klar ist wie viele Personen jemals bekannt sein werden, ist ein Array keine sinnvolle Struktur. Zudem verhindert `Set` automatisch das duplizieren von Bekannten.

c) Implementieren Sie die Methode `transitivelyKnows(Person p)` der Klasse `Person` welche prüft, ob die Person eine übergebene Person `p` kennt, oder ob eine der Bekannten sie kennt.
Hinweis: *Achten Sie auf Zyklen in den Bekanntenkreisen!*
Hinweis: *Objektidentität genügt als Vergleich.*

Lösung

```java
public boolean transitivelyKnows(Person p) {
    this.transitivelyKnows(p, new LinkedList<>());
}
public boolean transitivelyKnows(Person p, List<Person> visited) {
    visited.add(this);
    if (this == p) {
        return true;
    } else {
        for (Person kp: knows) {
            if (kp == p) {
                return true;
            } else if (kp.transitivelyKnows(p, visited)) {
                return true;
            }
        }
    }
    return false;
}
```