Lösungen zu den Übungsaufgaben im Buch *Automaten und Sprachen: Theoretische Informatik für die Praxis* von Andreas Müller, ISBN 978-3-662-70145-4 (Softcover), ISBN 978-3-662-70146-1 (eBook), https://link.springer.com/book/10.1007/978-3-662-70146-1. Website zum Buch: https://autospr.ch

Kapitel 14: Programmiersprachen

14.1. Eine Tabellenkalkulation kann typischerweise nur mit Gleitkommazahlen rechnen und ist damit in der Stellenzahl auf weniger als 16 Stellen begrenzt. Konstruieren Sie ein Spreadsheet, welches zwei 25-stellige Dezimalzahlen addieren kann.

Lösung. Ein solches Spreadsheet kann konstruiert werden, indem man für jede Stelle der Summanden eine Zelle reserviert. Die ersten beiden Zeilen des Spreadsheets enthalten in den Zellen `A1`–`Z1` den ersten und in den Zellen `A2`–`Z2` den zweiten Summanden. In der dritten Zeile stehen die Überträge aus den Spalten rechts davon. In Zelle `Z3` wird dazu `0` eingetragen, in allen Zellen weiter links die Formel

```
=QUOTIENT(SUM(M1:M3);10)
```

(hier dargestellt für die Spalte `L`). In der vierten Zeile wird die Summe dargestellt, sie wird berechnet durch die Formel

```
=MOD(SUM(L1:L3);10)
```

(wieder für die Spalte `L`). Das Spreadsheet ist in Abbildung 1 dargestellt ○

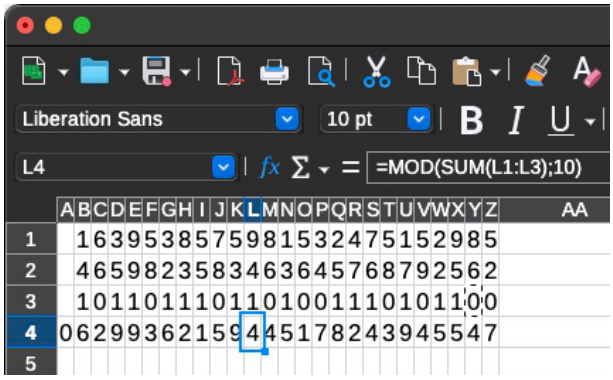
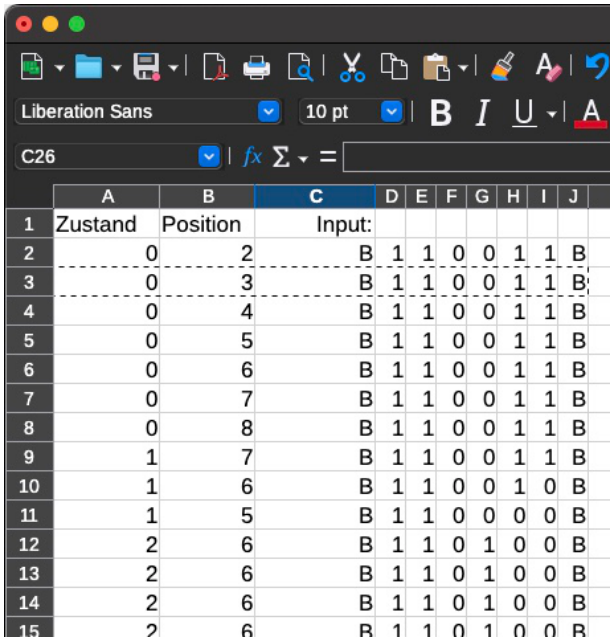


Abbildung 1: Spreadsheet zur Berechnung der Summe großer Dezimalzahlen

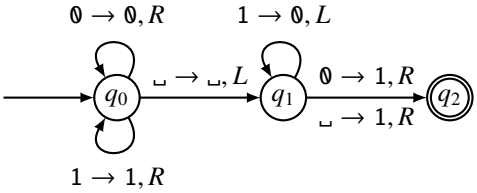
14.2. Das Zustandsdiagramm



Liberation Sans | 10 pt | C26

	A	B	C	D	E	F	G	H	I	J
1	Zustand	Position	Input:							
2	0	2	B	1	1	0	0	1	1	B
3	0	3	B	1	1	0	0	1	1	B
4	0	4	B	1	1	0	0	1	1	B
5	0	5	B	1	1	0	0	1	1	B
6	0	6	B	1	1	0	0	1	1	B
7	0	7	B	1	1	0	0	1	1	B
8	0	8	B	1	1	0	0	1	1	B
9	1	7	B	1	1	0	0	1	1	B
10	1	6	B	1	1	0	0	1	0	B
11	1	5	B	1	1	0	0	0	0	B
12	2	6	B	1	1	0	1	0	0	B
13	2	6	B	1	1	0	1	0	0	B
14	2	6	B	1	1	0	1	0	0	B
15	2	6	B	1	1	0	1	0	0	B

Abbildung 2: Spreadsheet zur Simulation einer Turing-Maschine, die eine Binärzahl inkrementiert.



beschreibt eine Turing-Maschine, die eine Binärzahl auf dem Band inkrementiert.

a) Konstruieren Sie ein Spreadsheet, welches die Berechnung schrittweise durchführen kann (siehe Abbildung 2). In dieser Implementation wird ein Blank auf dem Band durch den Buchstaben B dargestellt.

b) Überlegen Sie sich, ob Ihre Lösung sich auf einen Beweis für die Turing-Vollständigkeit des Spreadsheets verallgemeinern lässt.

Lösung. a) Wir verwenden die ersten beiden Spalten für den Zustand und die Adresse der aktuellen Kopfposition. In den Zellen der ersten und zweiten Spalte müssen daher Formeln eingegeben werden, die auf der Basis des aktuellen Bandzeichens und des Zustands in der ersten Spalte den neuen Zustand und die neue Position berechnen. Für die erste Spalte (in Zeile 6)

ist dies die Formel

```
=SWITCH(A5;
    0; SWITCH(INDEX(C5:J5; 1; B5);
        0; 0;
        1; 0;
        "B"; 1);
    1; SWITCH(INDEX(C5:J5; 1; B5);
        0; 2;
        1; 1;
        "B"; 2);
    2; 2)
```
(1)

und für die zweite Spalte

```
=SWITCH(A5;
    0; SWITCH(INDEX(C5:K5; 1; B5);
        0; 1;
        1; 1;
        "B"; -1);
    1; SWITCH(INDEX(C5:K5; 1; B5);
        0; 1;
        1; -1;
        "B"; 1);
    2; 0)+B5
```
(2)

Damit ist die Berechnung des neuen Zustands und der neuen Position abgeschlossen.

Für die Berechnung der neuen Zelleninhalte muss ein Formel eingegeben werden, welche nur aktiv wird, wenn die Spaltennummer mit der Kopfposition übereinstimmt. Die Formel

```
=IF($B5=COLUMN(G6)-2;
    SWITCH($A5;
        0; G5;
        1; SWITCH(G5;
            0;1;
            1;0;
            "B";1);
        2;G5);
    G5)
```
(3)

erreicht dies.

b) Es ist ziemlich offensichtlich, dass die drei Formeln (1), (2) und (3) sich direkt aus der Übergangsfunktion ableiten lassen. Für ein komplizierteres Zustandsdiagramm werden die Formeln komplizierter, aber nicht größer als die Definition der Turing-Maschine ist. Damit kann jedes beliebige Turing-Maschinenprogramm auch auf einem (vertikal unendlich ausgedehnten) Spreadsheet berechnet werden.

○

14.3. In der Sprache LOOP kann jede Variable direkt adressiert werden, und in der gleichen Instruktion können auch noch arithmetische Operationen ausgeführt werden. ‘Reinrassige’ RISC-CPUs

arbeiten anders. Sie verwenden eine Load-Store-Architektur, die arithmetische Operationen nur zwischen Registern zulässt. Das Laden eines Registers mit einem Wert aus dem Hauptspeicher ist immer getrennt von Rechenoperationen.

Betrachten Sie folgende RISC-LOOP genannte Modifikation von LOOP. Statt direkt adressierbarer Variablen $x_0, x_1, x_2, \ldots$ gibt es einen Index i, der auf jede der Variablen zeigen kann. Der Zeiger ist zu Beginn des Programms auf 0 initialisiert und kann inkrementiert oder dekrementiert werden, dazu gibt es zwei neue Befehle INCR und DECR, die immer auf i wirken. Der Index i kann jedoch nicht direkt auf einen bestimmten Wert gesetzt werden.

Mit x_i kann jeweils auf diejenige Variable zugegriffen werden, auf die i zeigt. x_i kann aber nicht direkt in einem arithmetischen Ausdruck verwendet werden, vielmehr muss ihr Wert zuerst in das Register r geladen werden, welches als einziges für arithmetische Operationen zur Verfügung steht. Das folgende Programm berechnet, was in LOOP die Anweisung $x_3 := x_2 + 4$ geschafft hat:

```
1:  INCR
2:  INCR
3:  r := x_i
4:  r := r + 4
5:  INCR
6:  x_i := r
```

Auch die LOOP Anweisung kann nicht mehr jede beliebige Variable als Argument verwenden, sondern nur den aktuellen Wert von r. Zeigen Sie, dass RISC-LOOP trotzdem nicht weniger leistungsfähig als LOOP ist.

Lösung. Der Zugriff auf Variablen kommt nur in Anweisungen der Form

$$x_j := x_i \pm c$$

und in der LOOP Anweisung

```
LOOP x_i DO ...END
```

vor. Um zu zeigen, dass RISC-LOOP nicht weniger leistungsfähig ist, müssen wir nur zeigen, dass wir jedes LOOP-Programm in ein RISC-LOOP-Programm umformen können. Dazu müssen wir offenbar ständig den Index i modifizieren, um auf die richtige Variable zugreifen zu können. Wir organisieren das so, dass wir dafür sorgen, dass nach einem Zugriff i immer wieder auf 0 zurückgesetzt wird. Die Zuweisung $x_3 := x_2 + 4$ ersetzen wir daher durch das Programm

```
1:  INCR
2:  INCR
3:  r := x_i
4:  DECR
5:  DECR
6:  r := r + 4
7:  INCR
8:  INCR
9:  INCR
10: x_i := r
11: DECR
12: DECR
13: DECR
```

Allgemein übersetzen wir $x_j := x_i \pm c$ durch

- i `INCR` Befehle
- Zugriff $r := x_i$
- i `DECR` Befehle
- Rechnung $r := r \pm c$
- j `INCR` Befehle
- Speicherung $x_j := r$
- j `DECR` Befehle

Analog übersetzen wir `LOOP`x_i durch

- i `INCR` Befehle
- Zugriff $r := x_i$
- i `DECR` Befehle
- Schleifenbefehl `LOOP` r.

Auf diese Weise lässt sich jedes `LOOP`-Programm in ein `RISC-LOOP`-Programm übersetzen, `RISC-LOOP` ist also mindestens so leistungsfähig wie `LOOP`. ○