Mathematik, Berechnung und Basteleien

Author:
Oliver Skoček

Contents

Einführung

Im Sommer 2019 setzte ich mir in den Kopf ein Projekt zu starten, über das ich schon sehr lange nachgedacht hatte. Der ursprüngliche Plan, der vielleicht noch umgesetzt wird, aber zur Zeit auf Eis liegt, war es eine Rechenmaschine wie in den 1930ern zu bauen, basierend auf elektromechanischen Bauteilen wie Relais und einem Lochstreifenlesegerät. Von der Idee diese Relais selbst zu bauen und zwar aus Parkett Holzleisten, Zimmermannsnägeln, Aluminiumfolie und lackiertem Kupferdraht gab ich etwa zur selben Zeit auf wie die Idee Relais zu verwenden. Dies geschah zum einen, weil ich frustriert war mit der Unzverlässigkeit der Schaltung und mit der hohen Stromstärke von etwa 0.7A, die für eine Schaltung in meinem Design notwendig war, und zum Anderen geschah es weil ich angefangen hatte mich mit Breadboards und integrierten Schaltkreisen zu beschäftigen.

Chapter 1

Aufbau einer Rechenmaschine

Was muss eine Rechenmaschine können, damit sie im Prinzip alles berechnen kann, auch wenn sie dafür eine sehr lange Zeit benötigt? Diese Frage wurde Mitte des zwanzigsten Jahrhunderts beantwortet. Mehrere Personen entwickelten unterschiedliche Definitionen des Berechenbaren, und im Laufe der Zeit konnte gezeigt werden, dass all iese hypotethischen Rechenmaschinen und Programmiersprachen gleichwertig sind. Die Gleichwertigkeit zwischen einer Rechenmaschine **A** und einer Rechenmaschine **B** ist gegeben, wenn die Rechenmaschine **A** die Rechenmaschine **B** simulieren kann und umgekehrt. Dies gilt unabhängig davon ob es sich um eine analoge Rechenmaschine, eine digitale Rechenmaschine, eine Quantenrechenmaschine oder einen Menschen mit Papier und Stift handelt.

Eine Rechenmaschine ist ein Apparat, der Listen von Befehlen abarbeitet und dabei drei fundamentale Bestandteile hat:

Register: Variablen, Platzhalter, Schmierzettel, et cetera. Ein Ding, dass ein Wort speichern kann, wobei dieses Wort von der Maschine gelesen werden kann und auch durch ein bestimmtes Wort ersetzt werden kann.

Grundoperationen: Eine Liste von grundlegenden Operation, welche die Maschine auf Wörtern, Paaren von Wörtern, et cetera ausführen kann. Die Ausgabe der Operation muss eindeutig sein und ist wiederum entweder ein Wort, ein Paar von Wörtern, et cetera. Die Eingabe Wörter liegen hierbei in bestimmten von der Operation abhängigen Registern und die Ausgabewörter werden wiederum in bestimmten operationsabhängigen Registern abgelegt.

Bedingte Verzweigung: Ein Weg wie der Ausgang einer Operation die Reihenfolge der abzuarbeitenden Befehle abändern kann. Diese Konstrukte

haben für gewöhnlich die Form: "Wiederhole bestimmte Teilliste von Befehlen, bis eine Bedingung erreicht ist"

Der zuletzt beschriebene Bestandteil einer Rechenmaschine, die **Bedingte Verzweigung** wird oft durch einen internen Zustand der Rechenmaschine realisiert, der für gewöhnlich über so genannte Flaggen implementiert wird. Dies sind Register mit nur einem Bit, die eine Aussage über den Ausgang der letzten Operation macht, welche die Rechenmaschine abgearbeitet hat. Diese Register sind die Ausnahme, sie sind nicht direkt in ihrem Wert setzbar, sondern nur indirekt und zusätzlich halten sie kein Wort, sondern genau einen Bit, also null falls die Aussage über die letzte Operation nicht zutrifft und Eins falls doch.

In den nächsten Kapiteln werden wir uns näher mit den Befehlen, die eine Rechenmaschine abarbeitet auseindandersetze. Zuerst soll hier aber noch der Unterschied und die Beziehung einer Rechenmaschine zum Konzept der Programmiersprache gezogen werden. Programmiersprachen sind Regelwerke nach denen Text geschrieben werden, der durch eine Rechenmaschine interpretiert und als Kette von Befehlen abgearbeitet werden kann. Typischerweise hat jede Rechenmaschine eine eigene interne Programmiersprache, welche die Maschinenbauteile interpretieren können und somit die Berechnung ausführen die wir uns wünschen.

Jede Programmiersprache besteht aus elementaren Bausteinen, Gundsymbolen oder Wörtern, die Variablen, Befehle, Operationen, Zuweisungen oder Verzweigungsanweisungen repräsentieren und aus Regeln, die festlegen welche Ketten dieser Grundsymbole Ausdrücke der Programmiersprache sind, also valide oder syntaktisch korrekte Programme. Ein Programm ist nichts anderes als ein Ausdruck, der die sprachlichen Regeln einer Programmiersprache erfüllt.

Synonym zum Begriff des Programms ist der Begriff des Algorithmus. Programmieren ist das entwerfen von Programmen. Jeder der noch nichts damit zu tun hatte, kann sich das komplett analog zu einem Koch vorstellen, der ein Rezept für einen unfähigen Lehrling schreibt. Das Rezept ist eine Abfolge an Operationen, die der Lehrling an der Küche ausführen muss, um ein Gericht herzustellen. Der Lehrling ist unfähig und deshalb muss man es ihm ganz genau aufschreiben, also präzise formuliert, nach gewissen Regeln, damit die Anweisung immer eindeutig ist und keine Interpretationsfreiheit zulässt. Genauso verhält es sich mit der Programmierung einer Rechenmaschine.

1.1 GOTO-Programme

Eine der einfachsten und am leichtesten verständlichen Arten von Programmiersprachen ist die Klasse der GOTO-Programmiersprachen. In der Praxis sind GOTO-Programmiersprachen zwar nur von geringer Bedeutung, da sich Programme, die in solchen Sprachen geschrieben wurden, wegen ihrer schwierigen Lesbarkeit, nur schwer warten lassen, jedoch sind die internen Maschinensprachen von allen Rechenmaschinen, die in der Praxis eingesetzt werden, GOTO-Programmiersprachen und (theoretisch interessant).

Es folgt eine Beschreibung einer sehr einfachen Form einer GOTO-Programmiersprache, die bereits alle Zutaten für eine universelle Programmiersprache enthält, und die wir fortan als Ausgangspunkt für unsere Diskussion des Berechenbaren ansehen.

Variablen: Variablen sind synonym zu verstehen mit Registern, es sind also Dinge die ein Wort speichern können, das gelesen werden kann, und durch ein anderes Wort ersetzt werden kann. Ein Wort wird in der folgenden Diskussion, um möglichst einfach zu bleiben, eine natürliche Zahl sein.

Definition 1.1. *Ein GOTO-Programm ist durch folgendes Schema definiert:*

1. *Das Kopieren des Wertes einer Variablen X auf eine Variable Y, kurz $Y = X$, ist ein GOTO-Programm.*
2. *Einer Variable X, einen bestimmten Wert A geben, kurz $X = A$[1], ist ein GOTO-Programm.*
3. *Einer Variablen Z die Summe zweier Variable X und Y als Wert zuordnen, kurz $Z = X + Y$, ist ein GOTO-Programm.*
4. *Einer Variablen Z die Differenz zweier Variable X und Y als Wert zuordnen, kurz $Z = X - Y$, ist ein GOTO-Programm.*[2]
5. *Das Stoppen des Programmes, kurz $HALT$ ist ein GOTO-Programm.*

[1] A ist zum Beispiel 12, daher kurz $X = 12$.

[2] Da wir mit natürlichen Zahlen und keinen ganzen Zahlen arbeiten, setzen wir eine Differenz, die in einer negativen Zahl resultiren würde auf den Wert Null.

6. *Seien W und V GOTO-Programme, dann ist*

```
W
V
```

ein GOTO-Programm. Daher zwei GOTO-Programme untereinandergeschrieben bilden ein GOTO-Programm. Praktisch bedeutet dies, dass zuerst das obere Programm und dann das untere Programm ausgeführt wird. Beispiel:

```
X = Y
Z = X + Y
```

Hier wird der Variable X der Wert der Variablen Y zugeordnet und anschließend wird der Variablen Z die Summe der Variablen X und Y zugeordnet.

7. *Das Springen zu der n-ten Zeile des GOTO-Programms, falls die Variable X einen Wert ungleich Null aufweist, und nichts tun falls es einen Wert gleich null hat, kurz IF X GOTO n.*
Beispiel (Arithmetische Multiplikation):

```
X = 0
X = X + Y
Z = Z - 1
IF Z GOTO 2
```

Das Beispielprogramm unter Punkt sieben beschreibt die Multiplikation zweier natürlicher Zahlen Z und Y, daher wenn das Programm zu Ende gelaufen ist, steht in der Variable X das Produkt der Werte, die zum Start des Programmes in der Variablen Z und Y gespeichert waren. Es ist zu beachten, dass wir eins basiert nummerieren. Also mit Zeilennummerierung als Orientierung sieht unser Programm so aus:

```
1: X = 0
2: X = X + Y
3: Z = Z - 1
4: IF Z GOTO 2
```

Wie nicht unschwer zu erkennen ist, steckt der Grund warum wir dies eine GOTO-Programmiersprache nennen im siebten Punkt der Definition. Dieser Punkt legt fest wie bedingte Verzweigung in der Programmiersprache funktioniert. Im Laufe der nächsten Kapitel werden wir weitere Möglichkeiten kennen lernen wie man dies bewerkstelligen kann.

Übungsbeispiel 1: Schreibe ein GOTO-Programm, dass die Fakultät einer natürlichen Zahl n berechnet, kurz $n!$. Die Fakultät ist rekursiv

definiert durch:

$$0! = 1$$

$$(n+1)! = (n+1) * n!$$

In einer einzigen Formel lässt sie sich aber auch, weniger präzise, folgendermaßen definieren:

$$n! = n * (n-1) * \cdots * 2 * 1$$

Beispiel: $5! = 5 * 4 * 3 * 2 * 1$

1.1.1 Grundoperationen

Die Grundoperationen unserer GOTO-Programmiersprache sind die Summe und die Differenz zweier natürlicher Zahlen. Alternativ gibt es aber auch noch andere mögliche Operationen, die in Kombination unsere Grundoperationen ausdrücken können und damit genauso Kandidaten für Grundoperationen sind. Hier wollen wir kurz ein paar Beispiele bringen.

Nachfolger und Vorgänger Alternativ zur Addition und Subtraktion kann man auch die einstellige Operation des Nachfolgers ($S(n) = n + 1$) und seiner Umkehroperation Vorgängers ($T(n) = n - 1$)[3] einer natürlichen Zahl verwenden. Offensichtlich lässt sich die Addition in diesen Operationen ausdrücken:

```
1: X = S(X)
2: Y = T(Y)
3: IF Y GOTO 1
```

Übungsbeispiel 2: Zeige wie sich die Subtraktion durch S und T in einem GOTO-Programm ausdrücken lässt.

Nachfolger und Gleichheit Die Vorgängeroperation lässt sich auch durch eine Gleichheitsrelation austauschen. Relationen sind Operationen, die als Resultat entweder Null für falsch oder Eins für wahr ausgeben. In unserem Fall nimmt die Operation zwei natürliche Zahlen X und Y und schreibt eine Eins, falls die Werte gleich sind, und sonst eine Null, in die Variable Z.
Als Operation schreiben wir dies als: $Z = (X == Y)$.

Übungsbeispiel 3: Zeige wie sich die Vorgängeroperation durch die Nachfolgeroperation und die Gleichheitsrelation in einem GOTO-Programm ausdrücken lässt.

[3] Es soll gelten $T(0) = 0$.

Übungsbeispiel 4: Zeige wie sich die Gleichheitsrelation in in der ursprünglichen Definition eine GOTO-Programms ausdrücken lässt.

1.2 WHILE-Programme

Eine, wegen ihrer leichteren Lesbarkeit, beliebtere Klasse von Programmiersprachen sind die WHILE-Programmiersprachen. Die meisten modernen Programmiersprachen sind unter anderem auch WHILE-Programmiersprachen.

Es folgt nun wie bei den GOTO-Programmen eine Beschreibung einer einfachen Form einer WHILE-Programmiersprache. Der einzige Untschied zu GOTO-Programmen liegt im siebenten Punkt der Definition, nämlich der Implementierung bedingter Verzweigungen.

Definition 1.2. *Ein WHILE-Programm ist durch folgendes Schema definiert (Übernehme Punkt 1-6 von der Definition eines GOTO-Programmes, lies einfach wo auch immer GOTO-Programm geschrieben steht, WHILE-Programm):*

7. *Wenn X ein WHILE-Programm ist und Z eine Variable, dann ist:*

```
WHILE Z:
    X
```

   auch ein WHILE-Programm. Es bedeutet, dass das WHILE-Programm X solange ausgeführt wird bis die Variable Z Null ist.
   Beispiel:(Arithmetische Multiplikation)

```
X = 0
WHILE Z:
    X = X + Y
    Z = Z - 1
```

   Wir sehen hier, wie bereits für GOTO-Programme gemacht ein WHILE-Programm, das das Produkt zweier natürlicher Zahlen X und Z berechnet.

An diesem einfachen Beispiel lässt sich wenn wir es mit dem zugehörigen GOTO-Programm vergleichen bereits erkennen, dass und inwiefern WHILE-Berechenbarkeit und GOTO-Berechenbarkeit equivalent sind, also sich zu jedem GOTO-Programm ein WHILE-Programm finden lässt und umgekehrt, dass daselbe berechnet. Was uns sogleich zum nächsten Abschnitt führt.

Übungsbeispiel 5: Mach Übungsbeispiel 1, 2 und 3 mit WHILE-Programmen, anstelle von GOTO-Programmen.

1.3 Church-Turing Hypothese

Die Church-Turing Hypothese (CT-Hypothese) ist eine nicht beweisbare Aussage über, dass was im Prinzip berechenbar ist. Inspiriert ist sie durch die Erkenntnis, dass alle Defintionen des Berechenbaren, also im Prinzip Programmiersprachen, zumindest die Berechnungen ausführen kann, die GOTO-Programme berechnen können. CT-Hypothese besagt also, dass jede Berechnung die im Prinzip möglich ist von einer Maschine ausgeführt werden kann, die GOTO-Programme verarbeiten kann. Hiermit ist auch klar weshalb die Aussage eher philosophisch und nicht beweisbar ist, da "das Berechenbare" nicht leicht fassbar/definierbar ist und wir heute nicht wissen können was morgen noch für Maschinen sein werden und welchen Gesetzen sie gehorchen werden. Es sei weiters noch angemerkt, dass wir aus ersichtlichem Grund die GOTO-Programmiersprache, sowie jede Programmiersprache, die jede Berechnung ausführen kann, die von einem GOTO-Programm ausgeführt werden kann, eine universelle Programmiersprache genannt werden soll. Eine solche universelle Programmiersprache ist sozusagen maximal in ihrer Fähigkeit Berechnungen auszuführen. Zur Motivation der Hypothese werden wir als nächstes beweisen, dass zu jedem GOTO-Programm ein WHILE-Programm existiert, dass daselbe berechnet und umgekehrt

IF THEN END Doch bevor wir dies zeigen soll eine bedingte Verzweigung eingeführt werden, die zwar nicht zwingend notwendig ist, daher wir können sie mit WHILE-Programmen sowie GOTO-Programmen bereits ausdrücken, aber sie werden, die in unseren Beweisen vorkommenden Programme kompakter und leichter lesbar machen, wenn wir sie verwenden.

Definition 1.3. *Sei Z eine Variable, P und Q sind GOTO/WHILE-Programme, dann nennen wir das Konstrukt*

```
IF Z THEN
        P
Q
```

eine IF THEN END Anweisung und sie bedeutet, dass falls Z ungleich Null ist, dann wird P ausgeführt und anschließend Q, andererseits falls Z gleich Null ist, dann überspringen wir P und führen gleich Q aus.

Übungsbeispiel 6: Schreibe ein GOTO-Programm, dass dieselbe Berechnung ausführt wie das oben beschriebene IF THEN END Konstrukt.

Übungsbeispiel 7: Schreibe ein WHILE-Programm, dass dieselbe Berechnung ausführt wie das oben beschriebene IF THEN END Konstrukt.

Übungsbeispiel 8: Schreibe ein WHILE-Programm, dass dieselbe Berechnung ausführt wie das oben beschriebene IF THEN END Konstrukt.

Übungsbeispiel 9: Zeige wie sich die Gleichheitsrelation als WHILE-Programm ausdrücken lässt.

Proof. **WHILE → GOTO:** Gegeben sei ein WHILE-Programm, wir können nun jedes Vorkommnis einer WHILE-Struktur

```
WHILE Z:
        P
```

nach folgendem Schema schrittweise durch eine GOTO-Struktur ersetzen:

```
n - 2: X = 1 - Z
n - 1: IF X GOTO m + 1
n    : P
m    : IF Z GOTO n
```

wobei n die Nummer des ersten Befehls im Programm P ist und die Rollen von X und Z so zu verstehen sind, dass zu jeder Variablen Z eine eigene Variable X erzeugt werden soll, die sonst nirgendwo verwendet wird.
Wenn dieser Prozess abgeschlossen ist, haben wir ein zu unserem ursprünglichen WHILE-Programm, equivalents GOTO-Programm erzeugt.

GOTO → WHILE: Gegeben sein ein GOTO-Programm P. Wir starten mit der Konstruktion eines equivalenten WHILE-Programmes Q indem wir die ersten beiden Zeilen schreiben:

```
Y = 1
WHILE Y
        X = 0
```

Wir rücken nun ein und der Rest des Programmes, dass wir konstruieren wird nun innerhalb dieser initialen WHILE-Anweisung laufen. Falls die Variable Y bereits in P vorkommt, wähle anstelle von Y eine Variable, die nicht in P vorkommt.

Anschließend fügen wir für jede Zeile im Programm P, der Reihe nach einen Segment folgender Form am unteren Ende von Q, mit der Einrückung der ersten Zeile nach dem ersten WHILE, hinzu.[4]

```
Z = (X == K)
IF Z THEN
        'Befehl von Programm P in der Zeile K'
        X = X + 1
```

Falls der Befehl in der K-ten Zeile von P eine Sprunganweisung ist, ersetzen wir den dort stehenden Befehl:[5]

[4] Da unsere WHILE-Programmiersprache keine IF THEN END Anweisungen enthalten, stehen die IF THEN END Anweisungen hier für ein funktionales equivalent in unserer WHILE-Programmiersprache.

[5] R steht hier für die im Befehl vorkommende Variable.

```
IF R GOTO n
```

durch

```
IF R THEN
        X = n
```

ansonsten schreiben wir den Befehl so wie er in P steht an die entsprechende Position. Das so erzeugte WHILE-Programm ist equivalent zum ursprünglichen GOTO-Programm. □

Abschließend sei noch erwähnt, dass die CT-Hypothese nichts über die Anzahl der Rechenschritte oder die Zeit, die eine Machine zur Berechnung benötigen wird aussagt. Ein Quantencomputer wird eine Faktorisierung einer großen natürlichen Zahl in kurzer Zeit vollbringen, während eine klassischer Computer Jahrhunderte braucht. Aber diesen Aspekt von unterschiedlichen Programmiersprachen und Maschinen haben wir bereits gesehen, als wir untschiedliche Grundoperationen für GOTO-Programme in betracht gezogen haben.

1.4 Kleenesche Normalform

Im vorangegangenen haben wir gesehen, wie zu jedem GOTO-Programm ein equivalentes WHILE-Programm erzeugt werden kann und umgekehrt. Wir modifizieren nun unsere WHILE-Programmiersprache, indem wir IF THEN END Anweisungen hinzufügen. Die resultierende Sprache nennen wir WHILE/IF-Programmiersprache. Sie ist equivalent zu unsere WHILE-Programmiersprache, da sich IF THEN END durch WHILE Anweisungen ausdrücken lässt, aber erlaubt nun die Konstruktion einer Kleenschen Normalform zu einem gegebenen WHILE-Programm.

Wir folgen nun dem Beweis aus dem letzten Abschnitt.
Gegeben ist ein WHILE-Programm P, wir wandeln es in ein equivalentes GOTO-Programm Q um, und konstruieren anschließend wie im Beweis ein equivalentes WHILE/IF Programm, dass nur mehr eine einzelne WHILE-Anweisung besitzt.

Beispiel: Gegeben sei folgendes WHILE-Programm, welches überprüft ob eine gegebene Zahl eine Primzahl ist.

Übungsbeispiel 10:

Chapter 2

Zahlensysteme

Bislang haben wir Programmiersprachen als etwas angesehen, dass Operationen auf natürlichen Zahlen ausführt. Bislang haben wir darauf verzichtet genauer darauf einzugehen, was genau eine natürliche Zahl ist und wie sie in einer physischen Rechenmaschine dargestellt werden können. Eine Rechenmaschine wird nie beliebig große natürliche Zahlen darstellen können, da sie immer limitiert sein wird in der Anzahl der wohlunterscheidbaren Zustände die sie annehmen kann.

Bisher sind wir nicht näher auf das Konzept der natürlichen Zahl eingegangen, sondern haben eine gewisses Verständnis desser vorausgesetzt. Hier wollen wir damit brechen und unsere Vorstellungen etwas konkretisieren. Fangen wir bei Null an, oder eigentlich bei Eins. Was sind die natürlichen Zahlen? Natürliche Zahlen werden verwendet zum Zählen. Schauen wir uns den Prozess des Zählens an.

2.1 Der Zählvorgang

Wir haben zwei Haufen wohlunterscheidbarer Objekte. Einen Haufen Birnen und einen Haufen Äpfel. Wenn wir wissen wollen ob genauso viele Birnen auf dem Birnenhaufen wie Äpfel auf dem Äpfelhaufen sind, können wir folgendermaßen vorgehen. Wir entfernen eine Birne vom Birnenhaufen und einen Apfel vom Apfelhaufen und tun dies so lange bis einer der Haufen verschwunden ist. Wenn beide gleichzeitig verschwinden, dann sind es gleich viele Birnen wie Äpfel, ansonsten gibt es mehr Birnen respektive Äpfel wenn der Äpfelhaufen beziehungsweise der Birnenhaufen zuerst
verschwunden ist. Durch diesen Prozess ist man in der Lage Anzahlen von allen möglichen Gegenständen durch Haufen von Äpfeln darzustellen. Haufen von Äpfeln sind eine mögliche Darstellung von natürlichen Zahlen. Ein beliebiger Apfelhaufen kann gebildet werden indem man mit einem Apfel startet und schrittweise weitere Äpfel hinzufügt. Wir wünschen uns jetzt

aber eine weniger verderbliche und kompaktere Darstellung von natürlichen Zahlen. Hierzu abstrahieren wir unsere Äpfelhaufen.

Definition 2.1. *Jede Konstruktion mit den hier beschriebenen vier Eigenschaften nennen wir eine Darstellung der natürlichen Zahlen.*

1. *Es gibt eine erste natürliche Zahl, wir nennen sie Eins.*
2. *Es gibt zu jeder natürlichen Zahl n, eine eindeutige nächste natürliche Zahl, den Nachfolger $S(n)$.*
3. *Alle natürlichen Zahlen werden durch mehrmaliges Nachfolger bilden aus der Eins konstruiert.*
4. *Der Nachfolger $S(n)$ einer natürlichen Zahl n unterscheidet sich von n und jeder natürlichen Zahl, die im Bildungsprozess von n, also startend bei 1, über alle Nachfolgerbildungen bis hin zu n, auftaucht.*

Proposition 2.1. *Beliebige zwei Darstellungen natürlicher Zahlen sind equivalent*[1]*.*

Proof. Gegeben sind zwei Darstellungen natürlicher Zahlen A und B. Wir assozieren nun die Eins von A , kurz 1_A, mit der Eins von B, kurz 1_B. Falls ein Element x von A mit einem Element y von B assoziert wird dann wird auch der Nachfolger von x bezüglich A mit dem Nachfolger von y bezüglich B assoziert. Nach Eigenschaft vier aus der Definition der Darstellung natürlicher Zahlen werden unterschiedlichen Elementen von A mit unterschiedliche Elemente von B assoziiert. Zusammen mit Eigenschaft drei folgt damit sofort, dass diese Assoziation ϕ eine umkehrbar eindeutige Abbildung ist, welche die Nachfolgerabbildung erhält. In Formeln können wir den Sachverhalt ausdrücken als:

$$\phi(S_A(x)) = S_B(\phi(x))$$

für beliebiges x in B, wobei ϕ die oben definierte Assoziation ist und S_A beziehungsweise S_B die Nachfolgeroperationen von A respektive B sind. □

2.2 Das dekadische Zahlensystem

In der Schule haben wir eine besonders effiziente Darstellung natürlicher Zahlen kennen gelernt, das dekadische Zahlensystem. Zeigen wir nun, dass diese tatsächlich natürliche Zahlen im oben beschriebenen Sinn sind.

[1] Equivalent bedeutet hier, dass es eine eindeutig umkehrbare Abbildung gibt, die mit den Nachfolgeroperationen der beiden Darstellungen verträglich ist.

Wir starten mit zehn Symbolen $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ und der Einfachheit halber starten wir nicht bei Eins, sondern bei Null. [2]

Definition 2.2. *Das dekadische Zahlensystem ist durch folgende Regeln definiert:*

1. *Die zehn Symbole haben eine feste Reihenfolge* $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$*. Diese Reihenfolge definiert eine Operation* u*, die ein Symbol nimmt und das nächste Symbol in der Reihenfolge ausgibt. Wenn es das Symbol* 9 *bekommt gibt es* 0 *aus.*
   Daher $u(0) = 1$*,* $u(1) = 2$*,* $u(2) = 3$*,* $u(3) = 4$*,* $u(4) = 5$*,* $u(5) = 6$*,* $u(6) = 7$*,* $u(7) = 8$*,* $u(8) = 9$*,* $u(9) = 0$*.*

2. *Eine dekadische Zahl ist eine endliche Abfolge dieser zehn Symbole. Hierbei schreiben/lesen wir von rechts nach links. Das erste Symbol oder wir sagen die erste Stelle ist das rechteste Symbol.*
   Beispiele:

   ```
   28420
     455
     390
      89
   ```

3. *Die Eins des dekadischen Zahlensystems ist:*

   ```
   1
   ```

4. *Wir definieren nun die Nachfolgeroperation des dekadischen Zahlensystems als ein Pseudo-GOTO-Programm.*

```
1: 'Starte bei der ersten Stelle.'
2: X = 'Symbol an der derzeitigen Stelle.'
3: X = u(X)
4: 'Setze den Wert der derzeitigen Stelle auf X.'
5: Z = X==0
6: if Z THEN
7:       'Gehe zur naechsten Stelle,
         falls keine Stelle mehr übrig ist
         füge eine Stelle mit dem Wert 0 hinzu
         und gehe zu dieser Stelle.'
8:       if Z GOTO 2
9: HALT
```

[2] Der Startpunkt ist irrelevant, solange es ein erstes Element gibt. Falls notwendig kann man einfach das erste Element nachträglich entfernen und man hat eine Struktur die bezüglich der ursprünglichen Struktur mit dem zweiten startet, aber die equivalent zur ursprünglichen Struktur ist.

Übungsbeispiel 11: Spiele den Algorithmus für einige Beispiele von Zahlen durch um dich mit der Nachfolgeroperation vertraut zu machen.

Corollary 2.3. *Die Stellen des dekadischen Zahlensystems sind selbst ein Zahlensystem.*

Übungsbeispiel 12: Beweise Corollary 1.6.

2.3 Das binäre Zahlensystem

Nachdem wir uns jetzt das dekadische Zahlensystem angeschaut haben, kommen wir nun zum eigentlichen Thema, dem dualen Zahlensystem oder binären Zahlen. Im binären Zahlensystem gibt es im Gegensatz zu den zehn Symbolen des dekadischen Zahlensystem nur zwei Symbole nämlich $0, 1$. Wie bei den dekadischen Zahlen legen wir eine Reihenfolge der Symbole fest, nämlich $0, 1$ und definieren das duale Zahlensystem analog zum dekadischen. Zur Wiederholung:

Definition 2.4. *Das dual/binäre Zahlensystem ist durch folgende Regeln definiert:*

1. *Die zwei Symbole haben eine feste Reihenfolge* $0, 1$.
2. *Eine binär Zahl ist eine endliche Abfolge dieser zwei Symbole. Hierbei schreiben/lesen wir von rechts nach links. Das erste Symbol oder wir sagen die erste Stelle ist das rechteste Symbol.*
   Beispiele:

   ```
   101010
      101
      111
       10
   ```

3. *Die Eins des dualen Zahlensystems ist:*

   ```
   1
   ```

4. *Wir definieren nun die Nachfolgeroperation des dualen Zahlensystems als eine Art GOTO-Programm:*

   ```
   1: 'Starte bei der ersten Stelle.'
   2: X = 'Symbol an der derzeitigen Stelle.'
   3: X = u(X)
   4: 'Setze den Wert der derzeitigen Stelle auf X.'
   5: Z = X==0
   6: if Z THEN
   ```

```
7:      'Gehe zur naechsten Stelle,
        falls keine Stelle mehr übrig ist
        füge eine Stelle mit dem Wert 0 hinzu
        und gehe zu dieser Stelle.'
8:      if Z GOTO 2
9: HALT
```

Beispiel: Alle vier stelligen binären Zahlen

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Übungsbeispiel 13: Spiele den Algorithmus für einige Beispiele von Zahlen durch um dich mit der Nachfolgeroperation vertraut zu machen.

Übungsbeispiel 14: Berechne die binäre Darstellung der dekadischen Zahlen 23, 12 und 1000. Finde eine schnelle Methode ohne alle Zahlen von 1 bis zu der gewissen Zahl durchzugehen.

Im Allgemeinen nennen wir Zahlensysteme wie das dekadische oder das duale Zahlensystem Stellenwertsysteme. Im Prinzip müssen wir nur ein paar wohlunterscheidbare Symbole aussuchen, eine Reihenfolge festlegen und wir können beliebige derartige Systeme festlegen. In der Informatik ist ein beliebtes Zahlensystem das Hexadezimalsystem. Hier haben wir die Symbole $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$.

Übungsbeispiel 15: Berechne die hexadezimale Darstellung der dekadischen Zahlen 15, 17 und 432.

In der Praxis hat sich das binäre Zahlensystem bewährt, da zum einen eine

enge Beziehung zwischen logischen Ausdrücken und dem Zahlensystem und zum anderen unsere Rechenmaschinen elektronisch sind und es praktisch einfacher ist einen Schaltkreis zu bauen, der zwei Zustände kennt, nämlich viel Spannung oder Stromstärke und sehr wenig Spannung oder Stromstärke und es genauso einfacher ist diese zwei Zustände für einen oder mehrere weiteren Schaltkreis zu unterscheiden. Es gab zwar einige Versuche von Rechenmaschinen mit nicht binärem Zahlen- system, und sogar analoge Rechenmaschinen, die direkt mit reelen Zahlen arbeiten, aber nichts davon konnte die Robustheit der binären Darstellung und die daraus resultierenden Vorteile schlagen.

Chapter 3

Aussagenlogik

Es ist nun an der Zeit, dass wir ein wenig Logik ins Spiel bringen. Logik ist die Lehre vom exakten Schließen; in einfacheren Worten beschäftigt sich die Logik damit wie man aus wahren Sätzen, wiederum wahre Sätze erzeugt. In diesem Kapitel werden wir uns mit Aussagenlogik oder Boolscher Logik befassen, doch bevor wir in die Tiefen der Logik starten, müssen wir zuerst ein paar Grundbegriffe definieren.

Eine **Aussage** ist ein sprachliches Konstrukt, dass entweder wahr oder falsch ist. Es muss hierbei prinzipiell möglich sein zu überprüfen ob die Aussage zutrifft, also wahr ist oder nicht. Nehmen wir die Aussage:

Die vierte Nachkommastelle von π ist 5.

Um zu Überprüfen ob diese Aussage wahr ist, müssen wir die vierte Nachkommastelle der Kreiszahl π berechnen und überprüfen ob der resultierende Wert gleich 5 ist. Was prinzipiell geht und praktisch möglich oder sinnvoll ist, ist oft verschieden, doch ist eine Diskussion dieses Themas hier fehl am Platz.

Man drückt den Sachverhalt, dass eine Aussage ϕ wahr ist beziehungsweise falsch ist durch den **Wahrheitswert** von ϕ (kurz $w(\phi)$) aus. Wir schreiben den Wahrheitswert 1, falls die Aussage wahr ist oder 0, falls die Aussage falsch ist. Präzise formulier heißt dies für den Wahrheitswert einer Aussage ϕ schreiben wir:

$$w(\phi) = \begin{cases} 1 & \text{falls } \phi \text{ wahr ist.} \\ 0 & \text{falls } \phi \text{ falsch ist.} \end{cases}$$

Logische Verknüpfung sind Operationen die eine bestimmte Anzahl von Aussagen nehmen und daraus eine neue Aussage produzieren, deren Wahrheitswert allein von den Wahrheitswerten der Aussagen aus denen sie produziert wurde abhängt.

Ein Beispiel für eine jedem bekannte logische Verknüpfung, die nur eine einzelne Aussage nimmt und daraus eine neue Aussage produziert, ist die **Negation** oder Verneinung (kurz $\neg$) einer Aussage.

$$\neg(\text{Der Himmel ist blau}) = \text{Der Himmel ist nicht blau.}$$

Hierbei gilt, dass die Verneinung die Wahrheitswerte umdreht, daher aus wahr mach falsch und aus falsch mach wahr.

$$w(\neg\phi) = 1 - w(\phi) = \begin{cases} 1 & \text{falls } w(\phi) = 0 \\ 0 & \text{falls } w(\phi) = 1 \end{cases} \tag{3.1}$$

Das logische **Und** ist eine logische Verknüpfung, die zwei Aussagen verbindet zu einer Aussage. Seien nun ϕ und ψ Aussagen, dann schreiben wir $\phi \wedge \psi$ für die durch das logische Und erzeugte Verbindung der Aussagen.

$$\phi = \text{Der Himmel ist blau.}$$

$$\psi = \text{Fische leben im Wasser.}$$

$$\phi \wedge \psi = \text{Der Himmel ist blau und Fische leben im Wasser.}$$

Diese Verknüpfung verhält sich genauso wie im gewohnten Sprachgebrauch $\phi \wedge \psi$ wahr, falls ϕ und ψ wahr sind und sonst falsch.

$$w(\phi \wedge \psi) = w(\phi) * w(\psi) = \begin{cases} 1 & \text{falls } w(\phi) = 1 \text{ und } w(\psi) = 1 \\ 0 & \text{sonst} \end{cases} \tag{3.2}$$

Abschließend kommen wir zum logische **Oder**, das wie das logische Und zwei Aussagen verbindet zu einer Aussage. Seien wieder ϕ und ψ Aussagen, dann schreiben wir $\phi \vee \psi$ für die durch das logische Oder erzeugte Verbindung der Aussagen.

$$\phi = \text{Der Himmel ist grün.}$$

$$\psi = \text{Fische leben im Wasser.}$$

$$\phi \vee \psi = \text{Der Himmel ist grün oder Fische leben im Wasser.}$$

Im Gegensatz zum Oder im gewöhnlichen Sprachgebrauch verhält sich das logische Oder jedoch anders. Das logische Oder ist wahr, sobald einer der beiden verbundenen Aussagen wahr ist.

$$w(\phi \vee \psi) = \begin{cases} 1 & \text{falls } w(\phi) = 1 \text{ oder } w(\psi) = 1 \\ 0 & \text{sonst} \end{cases} \tag{3.3}$$

Damit haben wir die wichtigsten logischen Verknüpfungen kennen gelernt und können damit bereits alles ausdrücken, was man in der Aussagenlogik ausdrücken kann. Wir werden später in diesem Kapitel noch weitere logische

Verknüpfungen besprechen, die eine besondere Erwähnung verdienen.

Als nächstes wollen wir konkretisieren was ein aussagelogisches System ist. Man startet mit elementaren Aussagen, so genannten **logischen Atomen**. Dies sind die Grundsymbole unseres Systems aus welchen wir zusammen mit den logischen Verknüpfungen alle möglichen Kombinationen bilden können. Wir können nun jeder dieser Kombinationen einen Wahrheitswert geben, indem wir einfach für jedes logische Atom einen Wahrheitswert fixieren. Erst durch diese Zuordnung werden unsere Symbolketten von logischen Verknüpfungen und Atomen eigentlich Aussagen mit definierten Wahrheitswerten. Solange man aber nur die Symbolketten betrachtet und die logischen Atome noch nicht mit Wahrheitswerten belegt hat, nennt man diese Konstrukte aussagenlogische Formeln.

Definition 3.1. *Aussagenlogische Formeln sind durch folgendes Schema definiert:*

1. *Die logischen Atome $\{\phi_1, \phi_2, \dots\}$, daher die Elemente einer Liste von Grundsymbolen sind aussagenlogische Formeln.*

2. *Wenn ϕ und ψ aussagenlogische Formeln sind, dann auch $\neg(\phi)$, $(\phi) \wedge (\psi)$ und $(\phi) \vee (\psi)$. Falls ϕ oder ψ logische Atome sind, darf man an der Stelle wo dies zutrifft die Klammern hier weglassen.*

3. *Jede aussagenlogische Formel wird aus den logischen Atomen und mehrmalige Kombination dieser durch logische Verknüpfungen erzeugt.*

Jede aussagenlogische Formel wird zusammen mit einer Belegung der logischen Atome mit Wahrheitswerten, eine Aussage, durch folgendes Prinzip.

Definition 3.2. *Sei β eine **Belegung** der logischen Atome $\{\phi_1, \phi_2, \dots\}$, daher eine Zuordnung von 0 oder 1 zu jedem logischen Atom, dann lassen sich durch β Wahrheitswerte w für beliebige aussagenlogische Formeln durch folgendes Schema berechnen:*

Sei ϕ eine aussagenlogische Formel, dann gilt

1. *falls ϕ ein logisches Atom ist setzen wir den Wahrheitswert*

$$w(\phi) = \beta(\phi) \tag{3.4}$$

2. *falls ϕ kein logisches Atom ist, muss es nach Konstruktion entweder die Negation $\neg$ angewendet auf eine aussagenlogische Formel p sein, oder eine Verknüpfung durch das logische Und $\wedge$ oder das Oder $\vee$ von zwei aussagenlogischen Formeln p und q sein.*
   Im ersten Fall der Negation setzen wir den Wahrheitswert:

$$w(\phi) = 1 - w(p) \tag{3.5}$$

Im zweiten Fall des logischen Unds, setzen wir den Wahrheitswert:

$$w(\phi) = w(p) * w(q) \tag{3.6}$$

Im dritten Fall des logischen Oders, setzen wir den Wahrheitswert:

$$w(\phi) = \begin{cases} 1 & \textit{falls } w(p) = 1 \textit{ oder } w(q) = 1 \\ 0 & \textit{sonst} \end{cases} \tag{3.7}$$

Falls p beziehungsweise p und q logische Atome sind, wenden wir Punkt 1. an und sind fertig. Falls nicht, wenden wir wiederholt Punkt 2. an, bis wir ausschließlich logische Atome erreicht haben und wenden dann Punkt 1. an.

Wir zeigen nun anhand von einem Beispiel einer aussagenlogischen Formeln wie dies funktioniert.

Beispiel: Wir haben drei logische Atome ϕ_1, ϕ_2 und ϕ_3, mit Wahrheitswerten $w(\phi_1) = 0$, $w(\phi_2) = 1$ und $w(\phi_3) = 1$.

$\phi = (\neg(\phi_1)) \wedge (((\phi_2) \vee \phi_3) \wedge (\neg(\phi_2)))$

Berechne den Wahrheitswert $w(\phi)$:

Wir sehen, dass ϕ eine Und Verknüpfung von $p = \neg(\phi_1)$ und $q = (\phi_2 \vee \phi_3) \wedge (\neg(\phi_2))$ ist. Daher berechnet sich der gesuchte Wert durch

$$w(\phi) = w(p) * w(q) \tag{3.8}$$

Berechnen wir nun den Wahrheitswert für p

$$w(p) = w(\neg(\phi_1)) = 1 - w(\phi_1) = 1 - 0 = 1$$

und um den Wahrheitswert von q zu berechnen, bemerken wir, dass q eine Und Verknüpfung von $s = \phi_2 \vee \phi_3$ und $t = \neg(\phi_1)$ ist.

$$w(q) = w(s) * w(t) \tag{3.9}$$

Es ist ein leichtes die Wahrheitswertes für s und t zu berechnen:

$$w(s) = w(\phi_2 \vee \phi_3) = \begin{cases} 1 & \text{falls } w(\phi_2) = 1 \text{ oder } w(\phi_3) = 1 \\ 0 & \text{sonst} \end{cases} = 1$$

$$w(t) = w(\neg(\phi_2)) = 1 - w(\phi_2) = 1 - 1 = 0$$

Als nächstes setzen wir $w(s)$ und $w(t)$ in (1.9) ein und erhalten:

$$w(q) = w(s) * w(t) = 1 * 0 = 0$$

Abschließend setzen wir noch $w(p)$ und $w(q)$ in (1.8) ein und erhalten:

$$w(\phi) = w(p) * w(q) = 1 * 0 = 0$$

Übungsbeispiel 16: Berechne den Wahrheitswert der logischen Formeln

$$(\neg(\phi_1)) \vee ((\neg(\phi_2)) \wedge (\phi_3))$$

$$(\neg(\neg(\phi_3))) \vee \phi_1$$

$$\phi_3 \wedge (\phi_1 \vee (\neg\phi_1))$$

3.1 Wahrheitstafeln

Wir kommen nun zu einer sehr nützlichen Werkzeug zur Behandlung von aussagenlogischen Formeln, den Wahrheitstafeln. Hier wird eine Tabelle mit allen möglichen Wahrheitswerten für die Atome der aussagenlogischen Formel gebildet und für jede dieser Kombinationen schreibt man in der letzten Spalte den Wahrheitswert den die Formel für diese Kombination ergeben würde.

Wahrheitstafeln sind somit ein allgemeines Format zur Darstellung beliebiger aussagenlogischer Verknüpfunen und eignen sich hervorragend zum Finden von Lösungen aussagenlogischer Erfüllbarkeitsprobleme, die wir gleich anschließend behandeln werden. Starten wir mit den Wahrheitstafeln der drei Grund- verknüpfungen Verneinung, Und und Oder.

ϕ	$\neg\phi$
0	1
1	0

ϕ	ψ	$\phi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1

ϕ	ψ	$\phi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1

Eine gute Methode um alle möglichen Kombinationen von Wahrheitswerten von K logischen Atomen zu erzeugen und keine zu vergessen, ist es einfach die K stelligen binären Zahlen von Null, also der binären Zahl, die aus K Nullen besteht, bis zur größten binären Zahl mit K-Stellen, nämlich jener, die nur aus Einsern besteht, aufzuschreiben.

Beispiel: Wir haben drei logische Atome ϕ_1, ϕ_2 und ϕ_3. Die Wahrheitstafel für die aussagenlogische Formel $(\neg\phi_1) \vee ((\neg\phi_3) \wedge \phi_2)$ ist:

ϕ_1	ϕ_2	ϕ_3	$(\neg\phi_1) \vee ((\neg\phi_3) \wedge \phi_2)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Übungsbeispiel 17: Stelle die Wahrheitstafeln zu beiden aussagenlogischen Formeln in Übungsbeispiel 16 auf.

Abschließend sollen in diesem Kapitel noch ein paar logische Verknüpfungen über ihre Wahrheitstabelle vorgestellt werden.

Die logische **Implikation** ist eine Verknüpfung zweier Aussagen ϕ und ψ, die ausdrücken soll, dass wenn ϕ wahr ist, auch ψ wahr sein muss. Wir schreiben hier $\phi \implies \psi$.

ϕ	ψ	$\phi \implies \psi$
0	0	1
0	1	1
1	0	0
1	1	1

Falls ϕ falsch ist, kann ψ falsch oder wahr sein und $\phi \implies \psi$ ist trotzdem wahr.

Übungsbeispiel 18: Zeige mit Hilfe von Wahrheitstafeln, dass $(\neg\psi) \implies (\neg\phi)$ dieselben Wahrheitswerte hat wie $\phi \implies \psi$.

Die logische **Äquivalenz** zweier Aussagen ϕ und ψ drückt aus, dass die Wahrheitswerte der Aussagen gleich sind, kurz $\phi \equiv \psi$.

ϕ	ψ	$\phi \equiv \psi$
0	0	1
0	1	0
1	0	0
1	1	1

Übungsbeispiel 19: Die Verneinung der logischen Äquivalenz ist das ausschließende Oder, oder auch **XOR** genannt. Schreibe die Wahrheitstafel der XOR Verknüpfung auf.

3.2 Erfüllbarkeitsprobleme und Gleichungen

Aussagenlogische Formeln lassen sich wie die Terme und Gleichungen der Algebra, die wir aus der Schule kennen interpretieren. Logische Atome sind, wenn keine Wahrheitswerte definiert sind, nichts anderes als Variablen, daher Symbole mit unbestimmten Werten. Die Werte sind im Fall von aussagenogischen Formeln entweder Null oder Eins.

Als Gleichheit verwendet man in der Logik die Äquivalenz; Diese verhält sich wie die Gleichheit in der Schulalgebra.

Definition 3.3. *Seien daher ϕ, ψ und γ beliebige aussagenlogische Formeln dann gilt:*

1. *Eine Formel ist mit sich selbst äquivalent*

   (3.10) $$\phi \equiv \phi$$

2. *Wenn für zwei Formeln gilt*

   (3.11) $$\phi \equiv \psi \text{ dann gilt auch } \psi \equiv \phi$$

3. *Wenn für drei Formeln gilt*

   (3.12) $$\phi \equiv \psi \text{ und } \psi \equiv \gamma \text{ dann gilt auch } \phi \equiv \gamma$$

Eigentlich ist es notwendig Klammern um die beiden Formeln zu setzen, die links und rechts vom $\equiv$ Symbol sind, aber solange die Äquivalenz als Gleichheit betrachten und nicht als Verknüpfung, werden wir die Klammern weglassen.

Eine erste Folgerung aus der Verwendung der Äquivalenz als Gleichheit zusammen mit dem Sachverhalt, dass zwei Formeln genau dann äqivalent sind, wenn sie dieselben Wahrheitswerte haben und der Wahrheitswert, den eine logische Verknüpfung ergibt nur von den Wahrheitswerten der verknüpften Aussagen abhängt.

Corollary 3.4. *Seien ϕ_1, ψ_1, ϕ_2 und ψ_2 aussagenlogische Formeln dann gilt.*

1. *Aus $\phi_1 \equiv \phi_2$ folgt $\neg(\phi_1) \equiv \neg(\phi_2)$.*
2. *Aus $\phi_1 \equiv \phi_2$ und $\psi_1 \equiv \psi_2$ folgt $\phi_1 \wedge \psi_1 \equiv \phi_2 \wedge \psi_2$.*
3. *Aus $\phi_1 \equiv \phi_2$ und $\psi_1 \equiv \psi_2$ folgt $\phi_1 \vee \psi_1 \equiv \phi_2 \vee \psi_2$.*

Da die Äquivalenz selbst auch eine logische Verknüpfung darstellt, führt dies natürlich dazu, dass eine aussagenlogische Gleichung, zugleich als Gleichung aber auch als Formel interpretierbar ist.

Wir nehmen daher Abstand vom Begriff Gleichungen und betrachten das so genannte Erfüllbarkeitsproblem Aussagenlogischer Formeln.
In einfachen Worten ist das Erfüllbarkeitsproblem einer Formel ψ, die Suche nach Wahrheitswerten, also Null oder Eins, die wenn sie in ψ für die logischen Atome eingesetzt werden Eins als Wahrheitswert ergeben.

Definition 3.5. *Sei ψ eine aussagenlogische Formel und β eine Belegung der logischen Atome in ψ, sodass der zu β gehörende Wahrheitswert für ψ Eins ist, dann nennen wir die Belegung β eine Lösung des Erfüllbarketisproblems für ψ. Die Suche nach einer solchen Belegung, nennen wir folglich das Erfüllbarkeitsproblem von ψ.*

Genauso wie sich in der Schulalgebra Formeln umformen lassen in andere gültige Formeln, so lassen sich auch aussagenlogische Formeln umformen in äquivalente Formeln. Wir nennen die Algebra, die mit ausagenlogischen Formeln verbunden ist auch Boolsche Algebra nach dem Mathematiker George Bool, der nach unseren Aufzeichnungen, der erste war, der sich mit diesem Thema befasste.

Aussagenlogische Formeln lasssen sich bezüglich der logischen Und und der logischen Oder Verknüpfung genauso umformen wie die Multiplikation und Addition in der Schulalgebra. Die Besonderheit der Boolschen Algebra ist, dass das diese beiden Verknüpfungen in Bezug auf die Rolle, die sie bei den Umformungen einnehmen, austauschbar sind.

Es gilt in der Boolschen Algebra die **Assoziativität**, sowohl für das Und als auch für das Oder. Assoziativität bedeutet, dass solange wir nur Und oder nur Oder Verknüpfungen in einer Verkettung haben, ist die Reihenfolge der Verknüpfungen egal beziehungsweise lassen sich die Klammern beliebig setzen.

Seien ψ, ϕ und γ aussagenlogische Formeln, dann gilt.

$$(\phi \vee \psi) \vee \gamma \equiv \phi \vee (\psi \vee \gamma) \tag{3.13}$$

$$(\phi \wedge \psi) \wedge \gamma \equiv \phi \wedge (\psi \wedge \gamma) \tag{3.14}$$

Übungsbeispiel 20: Beweise die Formeln (1.10) und (1.11) indem du die Wahrheitstafel der Formel links der Äquivalenz und rechts der Äquivalenz aufschreibst und dich vergewisserst, dass sie gleich sind (Behandle dabei ψ, ϕ und γ wie logische Atome).

Übungsbeispiel 21: Wir haben in (1.10) und (1.11) gesehen, dass die Reihenfolge der Klammerung für drei Formeln verknüpft durch das logische Und oder das logische Oder egal ist. Zeige, dass dies für beliebige Viele gilt.

Eine direkte Konsequenz der Definition des Wahrheitswertes für Und und Oder Verknüpfungen ist die **Kommutativität** der beiden Verknüpfungen. Daher der Sachverhalt, dass es egal ist in welcher Reihenfolge zwei Formeln durch ein Und oder Oder verknüpft werden.

Seien ψ und ϕ aussagenlogische Formeln, dann gilt.

$$\phi \vee \psi \equiv \psi \vee \phi \tag{3.15}$$

$$\phi \wedge \psi \equiv \psi \wedge \phi \tag{3.16}$$

Übungsbeispiel 22: Beweise die Formeln (1.12) und (1.13) anhand der Definition des Wahrheitswertes (1.9), oder mit Wahrheitstafeln.

Bislang haben wir Umformungen von Formeln betrachtet, die nur das logische Und oder nur das logische Oder betrachten. Wir kommen nun zu Umformungen von Kombinationen von Und und Oder. Analog zur Schulalgebra gilt in der Boolschen Algebra das Gesetz der **Distributivität**, also das "Herausheben von Ausdrücken". Zur Erinnerung in der Schulalgebra gilt $a * (b + c) = (a * b) + (a * c)$. Im Unterschied zur Schulalgebra sind aber in der Boolschen Algebra die beiden Verknüpfungen, also das logische Und und das logische Oder gleichberechtigt.

Seien ψ, ϕ und γ aussagenlogische Formeln, dann gilt.

$$\phi \vee (\psi \wedge \gamma) \equiv (\phi \vee \psi) \wedge (\phi \vee \gamma) \tag{3.17}$$

$$\phi \wedge (\psi \vee \gamma) \equiv (\phi \wedge \psi) \vee (\phi \wedge \gamma) \tag{3.18}$$

Übungsbeispiel 23: Überzeuge dich von (1.14) und (1.15), indem du die zugehörigen Wahrheitstafeln aufstellst.

Übungsbeispiel 24: Seien a, b, c und d logische Atome und sei

$$\phi = (a \vee b) \wedge (c \vee d)$$

$$\psi = (b \wedge d) \vee (a \wedge c) \vee (b \wedge c) \vee (a \wedge d)$$

Zeige, dass wenn du mit ϕ startest, durch schrittweise Umformung bei der Formel ψ ankommen kannst.

Bevor wir auf Umformungen eingehen, die zusätzlich zum logischen Und und Oder auch die Negation berücksichtigen, soll noch das Konzept der **Idempotenz** vorgestellt werden. Eine Formel durch Und oder Oder verknüpft mit sich selbst ist gleich sich selbst. Sei ψ eine aussagenlogische Formel, dann gilt.

$$\psi \wedge \psi \equiv \psi \tag{3.19}$$

$$\psi \vee \psi \equiv \psi \tag{3.20}$$

Diese Eigenschaft ist eine große Hilfe, wenn man lange komplexe Formeln vor sich hat und auf eine einfachere Form bringen will. Wir werden im folgenden Kapitel mehr nützliche Vereinfachungen dieser Art kennen lernen.

3.3 Umformungen der Verneinung

Bisher haben wir ausschließlich Umformungen bezüglich der logischen Verknüpfungen Und und Oder kennen gelernt. Die wichtigsten Umformungen an der die Negation beteiligt ist und eine Verbindung mit dem logischen Und und Oder herstellt sind die DeMorganschen Gesetze.

Die **DeMorganschen Gesetze** besagen vereinfacht gesagt, dass man die Negation aus einer Formel herausheben kann, wobei sich ein logisches Und in ein Oder umwandelt und ein logisches Oder in ein Und.

Seien nun ϕ und ψ aussagenlogische Formeln, dann gilt das erste DeMorgansche Gesetz.

$$\neg(\phi \vee \psi) \equiv (\neg(\phi)) \wedge (\neg(\psi)) \tag{3.21}$$

und das zweite DeMorgansche Gesetz.

$$\neg(\phi \wedge \psi) \equiv (\neg(\phi)) \vee (\neg(\psi)) \tag{3.22}$$

Übungsbeispiel 25: Vergewissere dich von der Gültigkeit von (1.21) und (1.22) indem du die Wahrheitstafeln der Formeln links und rechts des $\equiv$ aufstellst.

Um diesen Abschnitt abzuschließen seien noch zwei wichtige Methoden zur Vereinfachung von logischen Formeln erwähnt, nämlich die **Elimination**

der Doppelten Negation, die besagt, dass wenn in einer Formel zwei Negationen hintereinander vorkommen, beide gestrichen werden können,

$$\neg(\neg(\phi)) \equiv \phi \tag{3.23}$$

und das Konzept der Tautologie. Eine Tautologie ist eine Formel, die unter beliebiger Belegung der logischen Atome den Wahrheitswert Eins ergibt. Im Gegensatz dazu ist eine Kontradiktion eine Formel, die unter jeder Belegung Null als Wahrheitswert ergibt. Tautologien sind ein nützliches Konstrukt aus dem sich die gesamte Aussagenlogik konstruieren lässt wenn man so will. Als Methode zur Vereinfachung von aussagenlogischen Formeln, kann man sich folgende Sachverhalte zu Nutzen machen.

Lemma 3.6. *Sei ϕ eine aussagenlogische Formel und ψ eine Tautologie dann gilt.*

1. *$\neg(\psi)$ is eine Kontradiktion.(Die Umkehrung gilt auch)*
2. *$\phi \wedge \psi \equiv \phi$*
3. *$\phi \vee \psi$ ist eine Tautologie.*

Übungsbeispiel 26: Beweise den Satz der Elimination der Doppelten Negation (1.23) durch Aufstellen der Wahrheitstafeln.

Übungsbeispiel 27: Zeige durch Anwendung der DeMorganschen Gesetze auf Punkt 2 in Lemma 1.13, dass $\phi \wedge \psi$ eine Kontradiktion ist, wenn ψ eine Kontradiktion ist.

3.4 Die disjunktive Normalform

Wir haben bislang gelernt, was aussagenlogische Formeln sind, wie wir sie umformen und wie wir Formeln mit Wahrheitstafeln darstellen können. In diesem Abschnitt lernen wir, wie man zu beliebigen Wahrheitstafeln, eine aussagenlogische Formel konstruieren kann, welche die Wahrheitstafel erfüllt. Da es aber immer mehr als eine Formel gibt, welche dieselbe Wahrheitstafel hat, muss die zu einer Wahrheitstafel konstruierte Formel eine spezielle Form haben, eine spezielle Darstellung, eine so genannte Normalform.

Eine Wahrheitstafel der Länge n ist, wie wir im Abschnitt Wahrheitstafeln gesehen haben, eine tabellarische Darstellung einer eindeutigen Zuordnung von Null oder Eins zu beliebigen Folgen von Null und Eins fester Länge n. In anderen Worten eine Tabelle in der links alle möglichen Kombinationen von Null und Eins der Länge n einmal stehen und rechts der eindeutige zugeordnete Wert, Null oder Eins.

Hier ein Beispiel einer Wahrheitstafel der Länge 3.

0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Welcher aussagenlogischen Formel von drei logischen Atomen, nennen wir die Atome ϕ_1, ϕ_2 und ϕ_3, entspricht diese Wahrheitstafel?

Starten wir damit die Frage zu beantworten, indem wir für jeden Einser auf der rechten Seite eine aussagenlogische Formel konstruieren, die genau dann wahr ist, wenn die korrespondierende Kombination aus Nullen und Einsen links erfüllt ist. Wir haben vier Einser links und wir nennen die unbekannten Formeln, die noch zu konstruieren sind ψ_1, ψ_2, ψ_3 und ψ_4.

Es sollen also folgende Wahrheitstafeln für ψ_1 bis ψ_4 gelten.

ϕ_1	ϕ_2	ϕ_3	ψ_1	ϕ_1	ϕ_2	ϕ_3	ψ_2	ϕ_1	ϕ_2	ϕ_3	ψ_3	ϕ_1	ϕ_2	ϕ_3	ψ_4
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	1	0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0	1	1	0	0	1	1	0	0	1	1	1	0	1	1	0
1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1
1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0

Wenn wir nun

$$\psi = \psi_1 \vee \psi_2 \vee \psi_3 \vee \psi_4 \tag{3.24}$$

betrachten, erschließt sich sofort, dass die zu ψ gehörende Wahrheitstafel eben die Wahrheitstafel erfüllt, die oben gegeben ist. Wegen diesem Verhalten nennt man das logische Oder auch "logische Addition".

Es fehlt also bloß Formeln zu finden für die ψ_1 bis ψ_4 stehen, also Formeln, welche die zugehörigen Eigenschaften erfüllen.

Fangen wir mit ψ_1 an. ψ_1 soll nur dann war sein, wenn die Atome ϕ_1, ϕ_2 und ϕ_3 alle den Wahrheitswert Null tragen. Andersherum betrachtet bedeutet

dies, dass die Verneinungen von ϕ_1 bis ϕ_3 alle Eins beziehungsweise wahr sein müssen damit die gesuchte Formel wahr ist.
Der einfachste Weg dies zu erreichen ist[1]

$$\psi_1 = (\neg(\phi_1)) \wedge (\neg(\phi_2)) \wedge (\neg(\phi_3)) \tag{3.25}$$

Fahren wir fort mit ψ_2; Wir verlangen von ψ_2, dass es genau dann wahr ist wenn ϕ_1 sowie ϕ_2 den Wahrheitswert Null tragen, also falsch sind, und ϕ_3 den Wahrheitswert Eins hat. Anders ausgedrückt ist ψ_2 genau dann Eins wenn die $\neg(\phi_1)$, $\neg(\phi_2)$ und ϕ_3 wahr sind.
Der einfachste Weg ist wieder die Verknüpfung dieser drei Formeln durch Und Operationen.

$$\psi_2 = (\neg(\phi_1)) \wedge (\neg(\phi_2)) \wedge (\phi_3) \tag{3.26}$$

Übungsbeispiel 28: Stelle die Wahrheitstafel der rechten Seiten der Gleichungen (3.25) und (3.26) auf um dich zu vergewissern, dass wir tatsächlich ein gültiges ψ_1 beziehungsweise ψ_2 gefunden haben.

Nach diesen beiden Beispielen erkennt man bereits das allgemeine Muster. Wenn eine Formel nur für eine bestimmte Belegung seiner logischen Atome mit Nullen und Einsen wahr sein soll, dann schreibe für jedes Atom, dass von dieser Belegung den Wert Null bekommt die Verneinung dieses Atom und für jedes andere Atom, schreiben wir das Atom selbst auf. Am Ende verknüpfen wir alles durch logische Unds.

Zeigen wir dies noch einmal an der gesuchten Formel ψ_3. Sie soll nur unter der Belegung B

$$B(\phi_1) = 0,\ B(\phi_2) = 1,\ B(\phi_3) = 1 \tag{3.27}$$

wahr sein, beziehungsweise für die Kombination in der vieren Zeile auf der linken Seite der Wahrheitstafel oben in diesem Abschnitt.

Wir schreiben somit $\neg(\phi_1)$, weil $B(\phi_1) = 0$; wegen $B(\phi_2) = 1$ und $B(\phi_3) = 1$ schreiben wir auch ϕ_2 und ϕ_2. Am Ende verknüpfen wir alles 'geschriebene':

$$\psi_3 = (\neg(\phi_1)) \wedge (\phi_2) \wedge (\phi_3) \tag{3.28}$$

Übungsbeispiel 29: Berechne ψ_3 nach dem oben beschriebenen Schema und schreibe die vollständige Formel für ψ auf.

[1]Man beachte, dass auf Klammern verzichtet wurde, da wir ja nun wissen dass die Klammerung wenn wir nur Und oder nur Oder haben egal ist, und deshalb eindeutige Lesbarkeit nicht notwendig ist.

3.4.1 NAND und NOR

Wir schließen dieses Kapitel mit einer kurzen Diskussion zweier wichtiger logischer Operationen ab. Die Besonderheit dieser Operationen ist, dass für beide gilt, dass sich jede mögliche logische Operation durch Komposition darstellen lässt.

Die erste dieser Operationen ist die Verneinung des Unds, auch **NAND**, aus dem Englischen 'not and'. Als logische Formel können wir die Operation schreiben als $\neg(\phi \wedge \psi)$ oder über die DeMorgansche Gesetze umgeformt $(\neg(\phi)) \vee (\neg(\psi))$. Die zugehörige Wahrheitstafel ist

ϕ	ψ	ψ_4
0	0	1
0	1	1
1	0	1
1	1	0

Übungsbeispiel 30: Konstruiere die disjunktive Normalform zu dieser Wahrheitstafel.

Übungsbeispiel 31: Zeige, dass die beiden Formeln äquivalent sind, indem du zeigst das beide die obige Wahrheitstafel als zugehörige Wahrheitstafel haben.

Theorem 3.7. *Jede logische Formel und somit jede mögliche Wahrheitstafel, lässt sich durch Komposition logischer Atome durch NAND Operationen darstellen.*

Proof. Nachdem wir bereits über die Disjunktive Normalform gesehen haben, wie sich beliebige Wahrheitstafeln durch Kompositionen des logischen Unds, Oders und der Verneinung, bleibt zu zeigen, dass sich das Und, das Oder und die Verneinung als Komposition von NAND Operationen darstellen lässt. Seien ϕ und ψ logische Atome dann zeigen wir zuerst, dass sich die Verneinung darstellen lässt:

$$\neg(\phi) \equiv \neg(\phi \wedge \phi) \equiv (\phi)\text{NAND}(\phi) \tag{3.29}$$

Nun kommen wir zum Und. Durch Anwendung der doppleten Verneinung Umformung ist schnell gezeigt, dass

$$\phi \wedge \psi \equiv \neg(\neg(\phi \wedge \psi)) \equiv \neg((\phi)\text{NAND}(\psi)) \tag{3.30}$$

Da wir bereits wissen wie sich die Verneinung darstellen lässt, haben wir schon jetzt gezeigt, dass sich das Und darstellen lässt. Es fehlt noch das

Oder. Wir fangen wieder an mit doppelter Verneinung, dann DeMorgan und kommen auf

$$\phi \vee \psi \equiv \neg(\neg(\phi \vee \psi)) \equiv \neg((\neg(\phi)) \wedge (\neg(\psi))) \equiv (\neg(\phi))\text{NAND}(\neg(\psi)) \tag{3.31}$$

□

Es ist nun eine leichte Übung zu zeigen, dass das **NOR**, also die Verneinung des Oders, in Formel $\neg(\phi \vee \psi)$ dieselbe Eigenschaften hat.

Übungsbeispiel 32: Zeige Dies.

Chapter 4

Elektronik

Chapter 5

Ein möglichst einfacher Digitalrechner

Ein Digitalrechner hat zwei Buchstaben, nämlich die Null und die Eins, aber zusätzlich hat jeder eine meistens fixe Wortgröße, die in der Anzahl der Stellen , der Bits beziehungsweise Buchstaben, also der Nullen und der Einsen, welche die Maschine als ein Wort betrachtet, gemessen wird. Dieses Wort ist das eigentliche Elementare Objekt der Rechenmaschine, jede Operation wird nicht auf einem einzelnen Bit, also auf einer Stelle des Wortes, sondern immer auf dem gesamten Wort ausgeführt. Genauso holen wir wenn wir den Inhalt einer Speicheraddresse zum Rechenkern holen immer ein ganzes Wort, dass dort steht und nicht einen einzelnen Bit. Ein Wort kann dabei für einen Buchstaben stehen, für eine Zahl oder die Addresse eines anderen Wortes.

5.1 Eine minimale Arithmetisch logische Einheit

Die Idee einer digitalen Rechenmaschine mit der kleinst möglichen Anzahl an arithmetischen und logischen Operationen, die in Kombination eine universelle Rechenmaschine ergeben, hat mich fasziniert, seit ich mir Gedanken über den Bau von Rechenmaschinen gemacht habe. Solche Minimal Konstruktionen sind in der Regel nur in der Theorie interessant, da sie natürlich mehr Rechenschritte benötigen als Maschinen mit mehreren Rechenoperationen. Diese zusätzlichen Rechenoperationen sind, zwar redundant was für bestimmte Menschen ein Schönheitsfehler sein kann, aber Schnelligkeit und praktikabilität sind in der echten Welt wichtiger.

Meine arithmetisch logische Einheit hat die folgenden zwei Operationen:

NAND: Die verneinte-Und Operation, die wie wahrscheinlich wie jedem Leser bekannt, durch verschiedene Kombinationen mit sich selbst, jede erdenkliche

boolsche Operation erzeugen kann. Somit lassen sich sämtliche logischen Funktionen mit dieser Operation ausdrücken. Zusätzlich setzt die NAND-Operation falls das NULL-Wort als Ergebnis erhalten wird ein Flag.

LSHIFT: Der links shift beziehungsweise die Linksverschiebung, bei der jede Stelle im Wort um einen bit nach links verschoben wird, hierbei wird die nullte Stelle auf Null gesetzt und die höchste Stelle geht in ein Flag über.

Um zu zeigen, dass diese Operationen ausreichen, müssen wir lediglich einen Algorithmus finden, der die Additions Operation mit diesen beiden Grundoperationen ausdrücken kann.

Bevor wir dies tun können müssen wir den Rest unserer Programmiersprache definieren. Meine Wahl fiel hierbei auf WHILE-Programme mit IF Verzweigungen. Insbesondere ließ ich mich einschränken durch die Tatsache, dass die Kontrolleinheit der Rechenmaschine selbst keine Additionsoperationen ausführen soll, da dies die Sinnhaftigkeit der Einschränkung auf die beiden Grundoperationen zu absurd scheinen lässt. Dies führte mich zur Kleenschen Normalform. Jedes WHILE-Programm, aber auch jedes GOTO- Programm lässt sich umschreiben sodass nur einer WHILE Schleife beziehungsweise GOTO Aufruf verwendet wird.
Die Kontrolleinheit muss in diesem Fall immer nur den jeweils nächsten Befehl ausspucken, oder im Fall einer IF Verzweigung einige Befehle überspringen und am Ende des Programms zurückspulen zum Anfang. Es ist weder die Eingabe einer absoluten Addresse noch einer relativen Addresse notwendig. Die Kontrolleinheit muss also nicht Addressen aus den Befehlen extrahieren und dem Befehlzeiger setzten, noch einen Teil des Befehls auf den derzeitigen Befehlszeiger draufaddieren. Dies führt zusätzlich dazu, dass es die Absurdität des Vorhabens nicht zu offensichtlich macht, dazu, dass unsere Befehle nicht allzu lang sein müssen. In unserem Fall wird ein Befehl ein Byte sein, wobei der Befehlsraum hierbei nicht ausgelastet sein wird.

Die Realisierung der Kontrolleinheit kann nun ein Lochstreifenlesegerät oder ein binär Counter zusammen mit einem 8 Bit Parallelspeicher (EEPROM oder FLASH) sein.

5.1.1 Die physische Realisierung

5.2 Der Hauptspeicher und Registerkarte

5.3 Die Sprache unserer Rechenmaschine

Unsere Rechenmaschine hat eine Wortlänge von vier Bit.[1] Die Architektur unserer Machine ist anglehnt an die Harvard Architektur, mit getrenntem Befehlsspeicher und Datenspeicher.
Jeder Befehl ist einen Byte lang und hat die Form:

$$(b_0, b_1, b_3, a_0, a_1, a_2, a_3, F)$$

Die Befehle, die unsere Rechenmaschine kennt sind die Folgenden:

HALT: $(b_0, b_1, b_2) = (0, 0, 0)$ Der Zustand der Rechenmaschine ändert sich nicht und kein weiterer Befehl wird mehr ausgeführt.

NAND + 3 bit Addresse: $(b_0, b_1, b_2) = (0, 0, 1)$ Berechnet die NAND Operation des Wortes an der Addresse $(a_0, a_1, a_2, 0)$ mit dem Wort an der Stelle $(a_0, a_1, a_2, 1)$ und schreibt das Ergebnis in die Registerkarte. Falls das Ergebnis der Operation Null ist wird das Flag auf Eins gesetzt, sonst auf Null.

LSHIFT + 3 bit Addresse: $(b_0, b_1, b_2) = (0, 1, 0)$ Wendet die LSHIFT auf das Wort an der Stelle $(a_0, a_1, a_2, 0)$ and und schreibt das Ergebnis in die Registerkarte. Hierbei wird das nullte Bit des Wortes auf Null gesetzt und der Wert des dritten Bits wird auf das Flag übertragen.

POP + 4 bit Addresse: $(b_0, b_1, b_2) = (0, 1, 1)$ Schreibt das Wort an der Addresse (a_0, a_1, a_2, a_3) in die Registerkarte.

PUSH + 4 bit Addresse: $(b_0, b_1, b_2) = (1, 0, 0)$ Schreibt das Wort in der Registerkarte an die Addresse (a_0, a_1, a_2, a_3).

LOAD + 4 bit Wort: $(b_0, b_1, b_2) = (1, 0, 1)$ Schreibt das Wort (a_0, a_1, a_2, a_3) in die Registerkarte.

IF: $(b_0, b_1, b_2) = (1, 1, 0)$ Wenn das Flag auf Eins steht wird der nächste Befehl in der Reihe als nächstes ausgeführt, ansonsten wenn das Flag auf Null steht wird der nächste Befehl ausgeführt dessen letztes Bit, der F Bit, Eins ist.

[1] Im Fachjargon nennt man dies auch einen Nibble.

RESET IF: $(b_0, b_1, b_2) = (1, 1, 1)$ Wenn das Flag Eins ist wird der Zähler auf Null gesetzt, sonst passiert nichts und der nächste Befehl in der Reihe wird ausgeführt.

Nun sind wir in der Lage die Additionsoperation in der Maschinensprache unserer Rechenmaschine auszudrücken.

5.4 Die Kontrolleinheit