UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

# Simulation und Konstruktion von Busy-Beaver-Maschinen

*Simulation and Construction of Busy Beaver machines*

**Bachelorarbeit**

verfasst am
**Institut für Theoretische Informatik**

im Rahmen des Studiengangs
**Informatik**
der Universität zu Lübeck

vorgelegt von
**Marvin Detzkeit**

ausgegeben und betreut von
**Prof. Dr. Till Tantau**

mit Unterstützung von
**Florian Chudigiewitsch**

Lübeck, den 15. August 2025

## Eidesstattliche Erklärung

*Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.*

_____

Marvin Detzkeit

## Zusammenfassung

Die Busy-Beaver-Funktion $BB(n)$ ist eine nicht berechenbare Funktion, die eine Zahl $n$ auf die maximale Anzahl an Schritten abbildet, die eine haltende Turingmaschine mit $n$ Zuständen ausführen kann. Das Halteproblem für Turingmaschinen mit $n$ Zuständen und damit viele mathematische Sätze, lassen sich auf $BB(n)$ reduzieren. Die Reduktion eines Satzes auf $BB(n)$ kann durch die Konstruktion einer Turingmaschine mit $n$ Zuständen erfolgen, die genau dann anhält, wenn der Satz wahr oder falsch ist. Für die Konstruktion solcher Maschinen wurden im Rahmen der Arbeit die Programmiersprache *BusyC* und ein dazugehöriger Compiler entwickelt, der ein Programm in eine Maschine übersetzt. Mithilfe dieser Werkzeuge wurde der Satz „Es gibt keine ungerade perfekte Zahl.“ auf $BB(222)$ und die Goldbach-Vermutung auf $BB(290)$ reduziert. Ergänzend wurde ein Turingmaschinen-Simulator entwickelt, der im Idealfall eine Maschine schneller simuliert, als ein CPU-Takt pro Maschinenschritt.

## Abstract

The Busy Beaver function $BB(n)$ is a non-computable function that maps a number $n$ to the maximum number of steps a halting $n$-state Turing machine can take. The Halting problem for $n$-state Turing machines, and with it, many mathematical theorems, can be reduced to the value of $BB(n)$. The reduction of a theorem to $BB(n)$ can be shown by constructing an $n$-state machine that halts, if and only if, the theorem is either true or false. For the construction of such machines, the programming language *BusyC* and a compiler which translates a program into a machine were developed. These tools were used to reduce the theorem „There is no odd perfect number.“ to $BB(222)$ and the Goldbach conjecture to $BB(290)$. Furthermore a Turing machine simulator was developed. Under ideal conditions, it simulates a machine faster than one CPU cycle per machine step.

# Inhaltsverzeichnis

# 1

# Einleitung

Die Busy-Beaver-Funktion $BB : \mathbb{N} \to \mathbb{N}$ bildet eine Zahl $n \in \mathbb{N}$ auf die maximale Anzahl an Schritten ab, die eine haltende Turingmaschine mit $n$ Zuständen ausführen kann. Sie ist nicht berechenbar und wächst schneller als jede berechenbare Funktion. Kenntnis über einen Wert $BB(n)$ der Funktion kann dafür genutzt werden das Halteproblem für Turingmaschinen mit $n$ Zuständen zu entscheiden.

Es lassen sich Turingmaschinen konstruieren, die genau dann halten, wenn ein bestimmter mathematischer Satz wahr oder falsch ist. Kennt man den Wert $BB(n)$ für die Zustandszahl $n$ der erzeugten Maschine, lässt sich entscheiden, ob der Satz wahr oder falsch ist.

## Kontributionen

Im Rahmen der Arbeit wurden folgende Kontributionen geleistet:

- Entwicklung eines schnellen Turingmaschinen-Simulators, der im Idealfall $< 1$ Takt pro Maschinenschritt benötigt
- Entwurf der eigenen, primitiven Programmiersprache BusyC
- Entwicklung eines Compilers für BusyC, der ein Programm in eine Turingmaschine übersetzt
- Konstruktion einer Turingmaschine mit 222 Zuständen, die genau dann anhält, wenn es eine ungerade perfekte Zahl gibt
- Konstruktion einer Turingmaschine mit 290 Zuständen, die genau dann anhält, wenn die Goldbach-Vermutung falsch ist
- Reduktion der beiden Probleme auf die jeweiligen Werte von $BB(n)$

## Ähnliche Arbeiten

Eine ähnliche Arbeit ist *A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory* [9] von Adam Yedidia und Scott Aaronson. In der Arbeit wurde eine Turingmaschine mit 7 910 Zuständen konstruiert, die genau dann hält, wenn ZFC inkonsistent ist. Für die Konstruktion der Maschine wurden ein Programm in der, für die Arbeit entworfene

Sprache, *Laconic* geschrieben und mit einem Compiler zu einer Turingmaschine kompiliert[1]. In der Arbeit wurden außerdem zwei Maschinen konstruiert, die jeweils genau dann halten, wenn die Goldbach-Vermutung (4 888 Zustände) und die Riemann-Vermutung (5 372 Zustände) falsch sind.

Laconic ist eine „High-Level"-Sprache, die mehr Abstraktion bietet als BusyC. Dies macht die Programmierung in Laconic einfacher, jedoch sind die erzeugten Maschinen *deutlich* größer: Die mit BusyC erzeugte Maschine zur Goldbach-Vermutung ist um 94% kleiner, als die entsprechende Maschine, die mit Laconic erzeugt wurde.

## Struktur

Kapitel 2 führt die theoretischen Konzepte ein, die für das Verständnis der Arbeit notwendig sind. Dazu zählen die Turingmaschine, das Halteproblem und die Busy-Beaver–Funktion. Zum Schluss folgt ein kurzer Abschnitt zur Entwicklung in C. Kapitel 3 behandelt den entwickelten Simulator und zeigt die Techniken, die für die Optimierung genutzt wurden. Kapitel 4 gibt eine Einführung in die Programmiersprache BusyC. Kapitel 5 und Kapitel 6 befassen sich mit dem BusyC-Compiler: Ersteres erläutert das Parsing eines Programms, das Zweite zeigt wie aus einem Programm eine Maschine konstruiert wird. Kapitel 7 baut auf den vorherigen drei Kapiteln auf. Es zeigt die Entwicklung von BusyC–Programmen, die genau dann halten, wenn ein mathematischer Satz falsch ist. Mithilfe des Compiler werden diese Programme zu Maschinen kompiliert. Das Resultat ist die Reduktion der Probleme auf Werte von $BB(n)$.

[1] Die Kompilierung von Laconic ist komplexer als hier beschrieben und kann in der Arbeit nachgelesen werden.

# 2

# Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen für die Busy-Beaver-Funktion eingeführt: Die Turingmaschine, das Halteproblem und zuletzt die Funktion selbst. Anschließend folgt ein kurzer Abschnitt zur Entwicklung in C.

## 2.1 Die Turingmaschine

Für diese Arbeit wird die Turingmaschine auf die Busy-Beaver-Funktion zugeschnitten. Gute allgemeine Definitionen findet man zum Beispiel in *Introduction to the Theory of Computation* [8] und *Introduction to Theoretical Computer Science* [2].

Definition 2.1 (Turingmaschine). Eine *Turingmaschine* mit $n$ Zuständen ist ein Tupel

$$(Q, \Sigma, \delta, q_0, q_{halt})$$

mit:

- $Q = \{q_0, \dots, q_{n-1}\}$ endliche Menge der *nicht-haltenden* Zustände.
- $\Sigma = \{0, 1\}$ Bandalphabet mit Blanksymbol 0
- $q_0 \in Q$ Startzustand
- $q_{halt} \notin Q$ spezieller Haltezustand (wird *nicht* zu $n$ mitgezählt)
- Übergangsfunktion

$$\delta : Q \times \Sigma \to (Q \cup \{q_{halt}\}) \times \Sigma \times \{L, R\}$$

Die Maschine besitzt ein beidseitig unendliches Band, das überall mit dem Blanksymbol initialisiert ist. Die Bandzellen sind relativ zu einer beliebigen Bezugszelle nummeriert, die die Nummer 0 hat. Der Kopf der Maschine startet auf dieser Zelle.

Eine *Konfiguration* ist ein Tripel $(q, t, i)$, wobei

- $q \in Q \cup \{q_{halt}\}$ der aktuelle Zustand ist
- $t : \mathbb{Z} \to \Sigma$ die aktuelle Bandbelegung ist
- $i \in \mathbb{Z}$ die aktuelle Kopfposition ist, gemessen relativ zu Zelle 0

Ein *Berechnungsschritt* von $(q, t, i)$ nach $(q', t', i')$ erfolgt wie folgt:

1. Bestimme den aktuellen Leseinhalt $a = t(i)$
2. Wende die Übergangsfunktion an: $\delta(q, a) = (q', b, d)$, wobei $q'$ der Folgezustand ist, $b$ das zu schreibende Symbol und $d \in \{L, R\}$ die Bewegungsrichtung
3. Setze $t'(i) = b$ und $t'(j) = t(j)$ für alle $j \neq i$
4. $i' = i - 1$ falls $d = L$, bzw. $i' = i + 1$ falls $d = R$

Wenn $q = q_{halt}$ gilt, ist kein weiterer Berechnungsschritt definiert und die Maschine hält an.

Es gibt $(4(n+1))^{2n}$ Turingmaschinen mit $n$ Zuständen. Diese Zahl ergibt sich aus der Anzahl der möglichen Definitionen[2] für $\delta$.

Abbildung 2.1 zeigt eine Turingmaschine mit 3 Zuständen und einen Ausschnitt des Bandes in der Startkonfiguration. Jeder Kreis stellt einen Zustand der Maschine dar, wobei gerichtete Kanten die Übergangsfunktion visualisieren und den Folgezustand angeben. Die Beschriftung einer Kante hat die Form $a/b/d$ mit $a, b \in \Sigma$ und $d \in \{L, R\}$: Dabei steht $a$ für das gelesene Symbol, $b$ für das zu schreibende Symbol und $d$ für die Bewegungsrichtung des Kopfes ($L$ für einen Schritt nach links, $R$ für einen Schritt nach rechts).

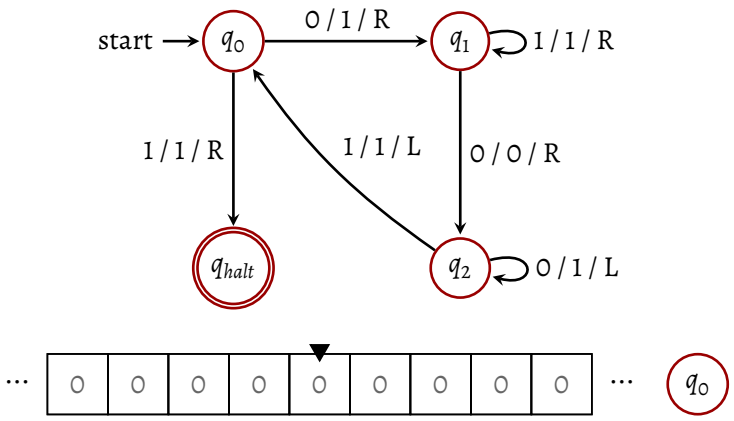



Abbildung 2.1: Eine Turingmaschine mit 3 Zuständen und das dazugehörige Band.

## 2.2 Das Halteproblem

Definition 2.2 (Halteproblem). Gegeben eine Turingmaschine $M$. Das *Halteproblem* ist die Frage, ob $M$ nach endlich vielen Berechnungsschritten den Zustand $q_{halt}$ erreicht.

Das Halteproblem ist für allgemeine Maschinen *unentscheidbar*. Beweise dazu gibt es in *Introduction to the Theory of Computation* [8] und *Introduction to Theoretical Computer Science* [2].

[2] $(|Q \cup \{q_{halt}\}| \cdot |\Sigma| \cdot |\{L, R\}|)^{|Q| \cdot |\Sigma|}$

## 2.3 Die Busy-Beaver-Funktion

Definition 2.3 (Busy-Beaver-Funktion). Die *Busy-Beaver-Funktion* $BB : \mathbb{N} \to \mathbb{N}$ ist definiert als

$$BB(n) = \max_{M \in H_n} \text{steps}(M),$$

wobei $H_n$ die Menge der haltenden Turingmaschinen mit $n$ Zuständen ist und $\text{steps}(M)$ die Anzahl der Berechnungsschritte angibt, die $M$ vor dem Halten ausführt.

Eine Turingmaschine mit $n$ Zuständen, die genau $BB(n)$ Schritte ausführt bevor sie anhält, wird *Busy Beaver* genannt. Die Funktion ist wohldefiniert, da es für jedes $n$ nur endlich viele Turingmaschinen mit $n$ Zuständen gibt und sich für jedes $n$ mindestens eine haltende Maschine konstruieren lässt, beispielsweise mit $\delta(q_0, 0) = (q_{halt}, 0, R)$.

Eingeführt wurde die Funktion in 1962 von Tibor Radó in seinem Paper *On Non-Computable Functions* [6], in dem er unter anderem zeigte, dass $BB(n)$ *nicht berechenbar* ist.

Die bislang bekannten Werte sind [1, 3]:

$$\begin{aligned} BB(1) &= 1, \\ BB(2) &= 6, \\ BB(3) &= 21, \\ BB(4) &= 107, \\ BB(5) &= 47\,176\,870. \end{aligned}$$

Für $n \geq 6$ sind keine Werte bekannt, jedoch gilt $BB(6) \geq 2 \uparrow\uparrow\uparrow 5$ [5]. Spätestens ab $n = 745$ lassen sich die Werte von $BB(n)$ nicht in ZFC beweisen [7].

Ist ein Wert $BB(n)$ der Funktion bekannt, ist das Halteproblem für Turingmaschinen mit $n$ Zuständen entscheidbar. Um zu entscheiden, ob eine Turingmaschine mit $n$ Zuständen hält, kann man diese für $BB(n)$ Berechnungsschritte simulieren. Falls sie nach so vielen Schritten noch nicht $q_{halt}$ erreicht hat, wird sie dies niemals tun.

Satz 2.4. *Es existiert keine berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(n) \geq BB(n)$ für alle $n \geq m$ mit $m \in \mathbb{N}$.*

*Beweis.* Angenommen es existiert eine solche Funktion $f$. Dann existiert eine weitere berechenbare Funktion $f' : \mathbb{N} \to \mathbb{N}$ mit $f'(n) \geq BB(n)$ für alle $n \in \mathbb{N}$. Mit $f'$ kann das Halteproblem entschieden werden, indem eine Turingmaschine mit $n$ Zuständen für $f'(n)$ Berechnungsschritte simuliert wird. Das ist ein Widerspruch zur Unentscheidbarkeit des Halteproblems, daher kann eine solche Funktion $f$ nicht existieren. □

Mit anderen Worten: $BB$ wächst schneller als jede berechenbare Funktion.

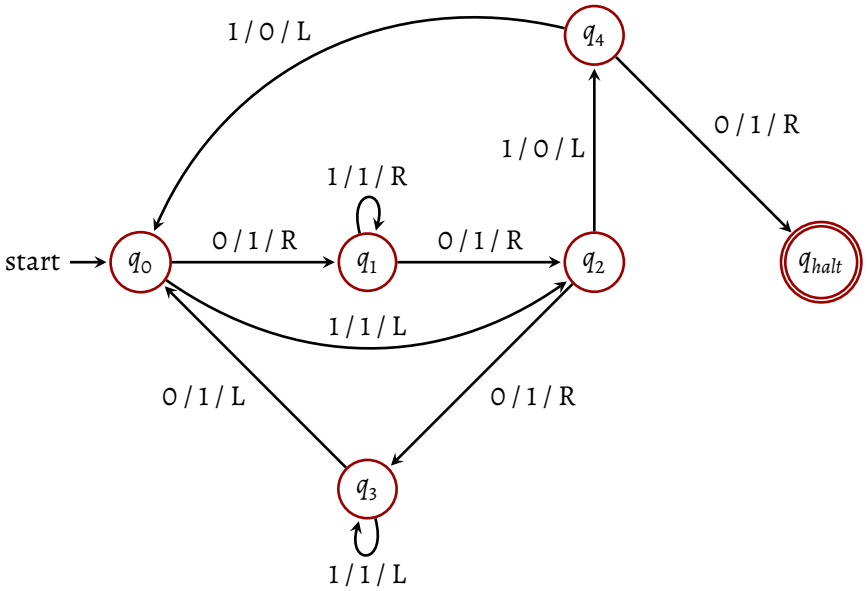



Abbildung 2.2: Ein Busy Beaver mit 5 Zuständen. Dieser Busy Beaver wird in Kapitel 3 als Benchmark für den Simulator genutzt.

## 2.4 Programmierung in C

Der Simulator und der Compiler, die für diese Arbeit entwickelt wurden, sind fast ausschließlich in *C* programmiert. Die Entscheidung C zu nutzen wurde wegen der Performance und persönlicher Präferenz getroffen. Es wird lediglich Python für das Einlesen der JSON-Dateien zum speichern von Turingmaschinen verwendet, weil die Standardbibliothek und die *Dictionary*-Datenstruktur das verarbeiten von JSON-Dateien in Python sehr angenehm macht.

Für das Kompilieren des Simulators und des Compilers wurde jeweils die *-O2*-Flag genutzt, welche bewirkt, dass das erzeugte Programm schneller läuft. Der Simulator ist mit der Flag fast doppelt so schnell!

Die Arbeit enthält einige C-Codeausschnitte. Der Code weicht teilweise von dem tatsächlich implementierten Code ab, jedoch ausschließlich in der Namensgebung von Bezeichnern. Dies dient dem Verständnis und der Einheitlichkeit. Unterschiede in der Programmlogik gibt es nicht.

# 3

# Simulator

Dieses Kapitel stellt den entwickelten Simulator mit dem Ziel möglichst „schnell“ zu laufen vor. Es werden die grobe Funktionsweise sowie die Techniken vorgestellt, die der Simulator nutzt um schnell zu laufen. Am Schluss gibt es einen Abschnitt mit Benchmarks.

## 3.1 Das Eingabeformat

Ein zentraler Teil des Simulators und später auch des Compilers ist das Format, in dem Turingmaschinen gespeichert werden. Diese liegen in JSON-Dateien vor. Jeder Zustand ist ein Schlüssel für ein Objekt, welches die *Kanteninformationen* des Zustands enthält. Diese beschreiben das Verhalten des Zustands je nach gelesenen Zeichen. Das folgende Beispiel zeigt einen Busy Beaver mit zwei Zuständen, dargestellt in diesem Format.

```
1   {
2       "q0":{
3           "blankWrite":1,
4           "blankShift":"r",
5           "blankState":"q1",
6           "oneWrite":1,
7           "oneShift":"l",
8           "oneState":"q1"
9       },
10      "q1":{
11          "blankWrite":1,
12          "blankShift":"l",
13          "blankState":"q0",
14          "oneWrite":1,
15          "oneShift":"r",
16          "oneState":"HALT"
17      }
18  }
```

*blankWrite* gibt das Zeichen an, das beim Lesen einer Null geschrieben wird, *blankShift* gibt die Schieberichtung des Kopfes beim Lesen einer Null an und *blankState* gibt den Zustand an, zu dem nach dem Lesen einer Null gewechselt wird. Entsprechend beschreiben

*oneWrite*, *oneShift* und *oneState* das Verhalten beim Lesen einer Eins. Der erste Zustand in der Datei ist der Startzustand der Maschine.

## 3.2 Die Repräsentation der Maschine in C

Die zu simulierende Maschine wird in C als Array des Structs *State* repräsentiert.

```
typedef struct{
    int id;
    int blankWrite;
    int blankShift;
    int blankState;
    int oneWrite;
    int oneShift;
    int oneState;
} State;
```

Die Variablen des Structs enthalten dieselben Informationen wie die gleichnamigen Schlüssel aus der JSON-Datei, mit dem Unterschied, dass alles als *int* kodiert ist. Null und Eins bleiben 0 und 1, die Schieberichtung links wird als $-1$, rechts als 1 kodiert und für den nächsten Zustand wird der Index des Zustands im Array verwendet. Die neue Variable *id* enthält den eigenen Index im Array.

Das Einlesen der JSON-Datei wird durch ein Python-Skript unterstützt. Der Grund dafür ist, dass Pythons Standardbibliothek erlaubt JSON-Dateien als *Dictionary* einzulesen, was die Verarbeitung sehr einfach macht. Das Skript liest die Datei ein und übersetzt sie in eine Form, die einfacher in C einzulesen ist.

## 3.3 Zusammenfassen von Zellen

Der Kern des Simulators ist das Zusammenfassen von Zellen zu einem Block bei der Simulation. Anstatt jeden einzelnen Schritt zu simulieren, wird das Verhalten der Maschine auf $k$ Zellen vorsimuliert, gespeichert und bei Bedarf ausgelesen.

### Der Speicher

Das Verhalten der Maschine auf einem Block hängt von den einzelnen Zellenwerten, dem Zustand der Maschine und der Position des Kopfes ab. Wir führen den Begriff *Konfiguration* für die Kombination aus Blockwert[3], Zustand und Kopfposition innerhalb des Blockes ein. Für die blockweise Simulation der Maschine sind nur die Konfigurationen relevant, bei denen sich der Kopf am Rand des Blocks befindet, weil erst wieder aus dem Speicher gelesen wird, wenn der Kopf den Block verlässt. Das Verhalten vor dem Verlassen des Blocks ist in der gespeicherten Konfiguration enthalten. Insgesamt gibt es demnach $2^k \cdot n \cdot 2$ Konfigurationen, die potentiell gespeichert werden müssen, wobei $n$ die Anzahl der Zustände der Maschine ist. Der Speicherbedarf ist exponentiell relativ zur Blockgröße.

[3] Ergibt sich aus den Werten der Zellen als Binärzahl interpretiert.

Es werden für jede gespeicherte Konfiguration die Folgenden Informationen gespeichert:

- Die Zellenwerte des Blocks nach der Simulation auf dem Block
- Der neue Zustand
- Zu welchem Block gewechselt wird nach der Simulation (+1 oder −1)
- Die Anzahl der Simulationsschritte, während der Simulation auf dem Block
- Die Differenz der Anzahl von Einsen auf dem Block vor und nach der Simulation

Die letzten beiden Informationen sind nicht nötig für die korrekte Simulation, sie sind aber interessant und werden deshalb gespeichert.

Diese Informationen werden in separaten Arrays gespeichert, die in einem Struct vereint sind. Es wurde während der Entwicklung getestet diese Arrays in ein einzelnes großes Struct-Array zusammenzufassen, jedoch war dieser Ansatz beim Testen messbar langsamer.

```
typedef struct {
    uint32_t* blockCache;
    State** stateCache;
    int* shiftDirectionCache;
    uint64_t* simulatedStepsCache;
    int* diffWrittenOnesCache;
} MachineCache;
```

Die einzelnen Arrays sind im Struct enthalten. Das Array *stateCache*, welches den nächsten Zustand speichert, ist kein 2D-Array sondern ein Array von Pointern. Für das Speichern des nächsten Zustands wird ein Pointer auf diesen Zustand im Maschinenarray gespeichert. Bei der Initialisierung der Simulation wird für alle Arrays speicher allokiert. Das *stateCache*-Array wird zusätzlich dazu genullt.

## 3.4 Die Repräsentation des Bandes

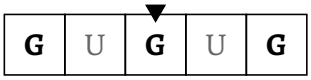
Das Band ist ein *uint32_t*-Array[4]. Die 32 Bit erlauben eine maximale Blockgröße von 32. Praktisch ist es nicht möglich die Blöcke so groß zu machen, da der Speicherbedarf zu groß ist. Die maximal mögliche Blockgröße beim Testen war 27 bei einer Turingmaschine mit 5 Zuständen. Die übrigen Bits werden auf Null gesetzt und ignoriert. Für Blockgrößen von 16 oder weniger bietet es sich an *uint16_t* oder sogar *uint8_t* zu nutzen, da dies den Speicherbedarf senkt. Vor dem Start der Simulation wird Speicher für das Band mit der Funktion *calloc* allokiert, welche das Band nullt.

### Wachsen des Bandes

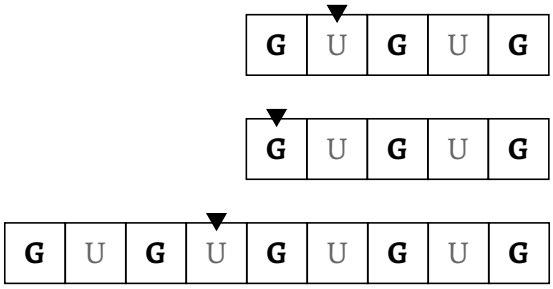
Um Verwirrung zu vermeiden wird der Begriff *Blockkopf* eingeführt. Der Blockkopf zeigt auf den aktuellen Block in der Simulation. Er ist für das „Blockband“, was der Kopf für das „Zellenband“ ist. Die initiale Größe des Bandes ist 50 001 Blöcke. Die Simulation beginnt

[4] Ein 32-Bit „unsigned int“, definiert in der Standard-Bibliothek von C.

mit dem Blockkopf in der Mitte. Für die Wahl der initialen Größe ist wichtig dass der Blockkopf um eine ungerade Anzahl verschoben werden muss, um das Band zu verlassen. Dies gilt, wenn von der Startposition des Blockkopfes aus in beide Richtungen eine gerade Zahl an Blöcken existieren bis zum Ende des Bandes. Es folgt eine Abbildung mit 5 Blöcken zur Demonstration.




Das Band wurde in Äquivalenzklassen aufgeteilt, die mit G und U beschriftet wurden. Die G-Blöcke sind die, auf denen der Blockkopf nach einer geraden Anzahl von Verschiebungen stehen kann, auf den U-Blöcken kann der Kopf nach einer ungeraden Anzahl von Verschiebungen stehen. Die G- und U-Blöcke wechseln sich ab, da der Blockkopf immer um genau einen Block verschoben wird. Daher gilt, dass der Blockkopf unmittelbar nach dem Verlasen des Bandes auf einem 1-Block steht. Der Simulator prüft daher nur in jeder zweiten Iteration, ob der Blockkopf noch auf dem Band steht. Das folgende ist eine Simulation des Blockkopfes beim Verlassen des Bandes.




Das Band wird um $b - 1$ Zellen erweitert, wobei $b$ die Ursprüngliche Bandlänge ist. Diese Darstellung zeigt das Erhalten der Eigenschaft, dass der Blockkopf nach verlassen des Bandes auf einem 1-Block steht.

## 3.5 Die Simulation

Die Simulation der Maschine findet in einer *while*-Schleife statt, in der die Werte für die aktuelle Konfiguration aus dem Speicher gelesen und verarbeitet werden.

Am Anfang der *while*-Schleife wird der Speicher-Index für die Konfiguration aus dem Blockwert, dem aktuellen Zustand und der letzten Schieberichtung berechnet. Beim Start der Simulation wird die letzte Schieberichtung auf rechts gesetzt, was bedeutet, dass der Kopf auf der linken Zelle des Startblocks steht. Der folgende Code zeigt die Funktion, die den Speicher-Index berechnet.

```
1 int getCacheIndexForCell(tapeType cell, int lastShiftDirection, int
    stateID) {
2     int cacheIndex;
3     cacheIndex = lookUpTable[lastShiftDirection];
4     cacheIndex += stateID * (1 << BLOCKSIZE);
5     cacheIndex += cell;
```

```
6     return cacheIndex;
7 }
```

Diese Funktion gibt jeder möglichen Konfiguration einen eigenen Index im Speicher. In der dritten Zeile wird die LookUp-Tabelle *lookUpTable* genutzt, um abhängig vom Wert von *lastShiftDirection* am Anfang oder in der Mitte des Speichers zu starten. *lastShiftDirection* ist immer entweder −1 oder 1. Da Arrays bei Index 0 beginnen, wird der folgende Trick verwendet:

```
1 placeholder[0] = 0;
2 placeholder[2] = cacheSize / 2;
3 lookUpTable = placeholder+1;
```

*placeholder* ist ein globales Array, das die benötigten Werte an den Stellen 0 und 2 enthält. *lookUpTable* ist ein Pointer, der mit der Adresse *placeholder*+1 initialisiert wird.

Als nächstes wird geprüft, ob die Daten für die aktuelle Konfiguration schon vorliegen oder noch vorberechnet werden müssen. Dafür wird der Pointer auf den nächsten Zustand gelesen. Falls der die Daten noch nicht vorliegen, hat der Pointer den Wert *NULL*, da das *stateCache*-Array bei der Initialisierung genullt wurde. Liegen die Daten schon vor, hat der Pointer nicht den Wert *NULL*[5].

Durch Vorberechnen des gesamten Speichers vor der Simulation könnte die Abfrage eingespart werden, jedoch müsste es stattdessen in jeder Iteration die Abfrage geben, ob die Maschine den Haltezustand erreicht hat und die Simulation abgebrochen werden muss. Beim „faulen" Ansatz, den Speicher erst bei Bedarf zu füllen, kann die Abfrage nach dem Haltezustand zu der Berechnung der Speicherwerte dazukommen. Das liegt daran, dass der Haltezustand sofort die Simulation abbricht, weshalb die Simulation nicht mehr fortgeführt wird, sobald der Haltezustand erreicht wird. Dadurch wird nur noch nach dem Haltezustand geprüft, wenn die Daten im Cache nicht vorliegen.

Das Vorberechnen des gesamten Speichers würde sehr viele unnötige Berechnungen beinhalten. Bei der Simulation des Busy Beavers von 5 mit Blockbreite 16 werden nur 106 Speicherwerte bei $2^{16} \cdot 5 \cdot 2 = 655\,360$ möglichen Konfigurationen berechnet.

Es folgt eine vereinfachte Version des Codes der *while*-Schleife, in welcher die Simulation stattfindet.

```
1  while (TRUE) {
2      int cacheIndex = getCacheIndex(block, lastShift, currentState->id);
3      nextState = machineCache.stateCache[cacheIndex];
4        if (nextState == NULL) {
5          // Berechnet Maschinenverhalten auf Block und schreibt in den
               Speicher
6          calculateBlockData(block, lastShift, currentState->id, cacheIndex);
7          nextState = machineCache.stateCache[cacheIndex];
8
9          // Prüfe ob Haltezustand erreicht wurde
10         if (newState < machine) break;
11     }
```

[5] Im C Standard wird festgelegt, dass die Adresse 0 ungültig ist.

```
12
13      // Lese Maschinenverhalten aus Speicher
14      *blockPointer = machineCache.blockCache[cacheIndex];
15      currentState = newState;
16      lastShift = machineCache.shiftDirectionCache[cacheIndex];
17      blockPointer += lastShift;
18      steps += machineCache.simulatedStepsCache[cacheIndex];
19      writtenOnes += machineCache.diffWrittenOnesCache[cacheIndex];
20
21      // Prüfe alle 2 Iterationen ob Blockkopf auf Band ist
22      if ((i & 1) && (((uintptr_t)blockPointer >= ((uintptr_t)(tape +
            tapeSize)))
23      || ((uintptr_t)blockPointer < (uintptr_t)tape))) growTape();
24  }
```

Nach dem Ausbrechen aus der Schleife wird noch ein letztes Mal das Maschinenverhalten aus dem Speicher gelesen, um die finale Anzahl an Schritten und Einsen auf dem Band zu erhalten.

## 3.6 Benchmarks

Als Benchmark für den Simulator wurde der Busy Beaver von 5 genutzt. Diese Maschine ist aus mehreren Gründen gut geeignet:

- Die Maschine läuft für viele Schritte (47 176 870)
- Für alle getesteten Blockgrößen wird in mehr als 99, 9% aller Iterationen der Speicher benutzt ohne die Maschine zu simulieren
- Die Anzahl der Zustände ist gering, was den Speicher verkleinert

Die Maschine wurde mit verschiedenen Blockgrößen und teilweise verschiedenen Bandtyp-Größen (8, 16 und 32 Bit) jeweils 107 mal simuliert. Die Ergebnisse werden in einem Box-Plot dargestellt. Die x-Achse ist mit „<Blockgröße>/<Bandtyp-Größe>“ beschriftet.

Der Benchmark wurde jeweils auf dem *T12*-Rechner des *Instituts für theoretische Informatik* und auf einem MacBook Pro 14 Zoll 2021 mit M1 Pro Prozessor ausgeführt.

### T12

| | |
|---|---|
| **Architektur** | x86_64 |
| **CPU** | AMD Ryzen Threadripper 3970X, 32 Kerne / 64 Threads |
| **Basistakt** | 2,20 GHz, Boost bis 4,55 GHz |
| **L1-Cache** | 1 MiB Daten (32× 32 KiB), 1 MiB Instruktionen (32× 32 KiB) |
| **L2-Cache** | 16 MiB (32× 512 KiB) |
| **L3-Cache** | 128 MiB (8× 16 MiB) |
| **RAM** | 251 GiB DDR4 |
| **OS** | Linux |

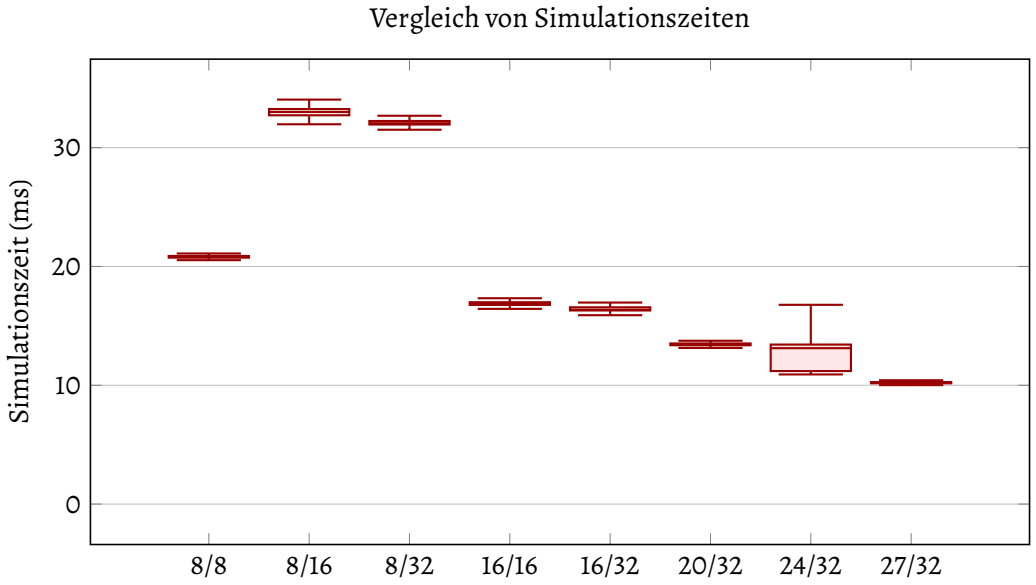



Abbildung 3.1: Ergebnisse auf dem T12

Die schnellste Simulation hat 9,966 ms gedauert und hat die Blockgröße 27 genutzt. Bei einer Taktfrequenz von 4,55 GHz sind das ca. 45,3 Millionen Takte für 47 176 870 Millionen simulierte Schritte. Im Durchschnitt wurde ein Maschinenschritt in ca. **0,96** Takten berechnet.

### M1 Pro

| | |
|---|---|
| **Architektur** | ARM64 (Apple Silicon) |
| **CPU** | Apple M1 Pro, 8 Kerne (6 Performance, 2 Effizienz) |
| **Basistakt** | dynamisch, bis ca. 3,22 GHz (P-Kerne) |
| **L1-Cache** | 192 KiB Instruktionen + 128 KiB Daten pro P-Kern,<br>128 KiB Instruktionen + 64 KiB Daten pro E-Kern |
| **L2-Cache** | 24 MiB für P-Kerne, 4 MiB für E-Kerne |
| **L3-Cache** | 16 MiB (gemeinsamer System-Cache für CPU und GPU) |
| **RAM** | 14,9 GiB |
| **OS** | macOS |

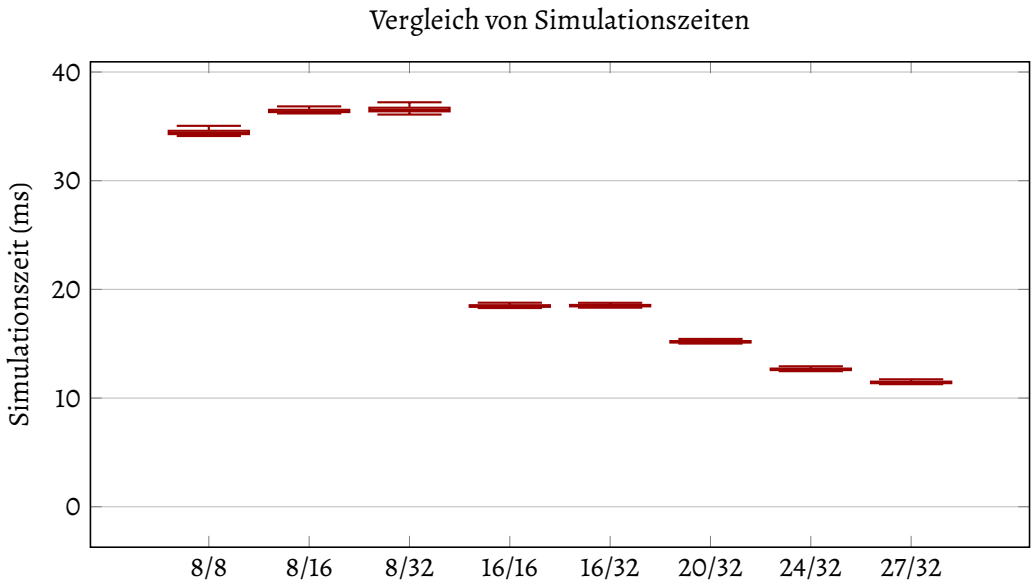



Abbildung 3.2: Ergebnisse auf dem M1 Pro

Die schnellste Simulation hat 11.289 ms gedauert und hat die Blockgröße 27 genutzt. Bei einer Taktfrequenz von 3, 22 GHz sind das ca. 36, 4 Millionen Takte für 47 176 870 Millionen simulierte Schritte. Im Durchschnitt wurde ein Maschinenschritt in ca. **0,77** Takten berechnet.

# 4

# Einführung in BusyC

In diesem Kapitel wird die Programmiersprache *BusyC* eingeführt. BusyC ist eine imperative Programmiersprache, die syntaktisch an C orientiert ist. Sie ist sehr primitiv, da sie darauf ausgelegt wurde, in eine Turingmaschine kompiliert zu werden. Für die Zwecke dieser Arbeit ist sie mächtig genug. Ein BusyC-Programm wird von oben nach unten ausgeführt und enthält keine Funktionen. Stattdessen müssen *Unterprogramm*, wie in Assembly, genutzt werden.

Abbildung 4.1: Das Logo der BusyC-Programmiersprache. Inspiriert vom C++-Logo.

## 4.1 Die Sprache

### Variablen

BusyC hat nur den Datentyp *uint* (unsigned integer) und erlaubt keine Arrays. Variablen müssen bei der Definition einen Initialwert zugewiesen bekommen. Sie können erst nach ihrer Definition benutzt werden.

Variablenbezeichner dürfen Buchstaben (a-z | A-Z) und Ziffern (0-9) enthalten . Folgende Einschränkungen gelten für Variablenbezeichner:

- Das erste Zeichen muss ein Kleinbuchstabe sein
- Variablenbezeichner müssen eindeutig sein
- Die Bezeichner „uint“, „goto“, „if“, „halt“ sind nicht gültig

Der folgende Codeschnipsel zeigt die Definition einer Variable mit dem Initialwert 5.

```
1 uint var = 5;
```

Die Initialisierung einer Variable findet **vor** der Laufzeit des Programms statt und nicht währenddessen. Diese Eigenschaft macht es möglich verwirrende Programme zu schreiben, indem man eine Variable mitten im Code definiert. Eine Variable kann ihren Wert nicht nochmal zugewiesen bekommen. Die Definition mit Initialisierung einer Variable ist kein Befehl in BusyC.

### Arithmetik

BusyC enthält die arithmetischen Operationen Inkrement und Dekrement. Letztere hat die Sondereigenschaft, dass bei Anwendung auf eine Variable mit dem Wert 0 dieser nicht verändert wird.

```
1 var++;
2 var--;
```

### Sprungbefehl

BusyC enthält den Befehl *goto*, mit dem zu einem *Label* gesprungen werden kann. Labels können überall im Programm platziert werden. Labelbezeichner können aus Großbuchstaben (A-Z), Ziffern (0-9) und Unterstrichen („_“) bestehen, wobei das erste Zeichen ein Großbuchstabe sein muss. Im Gegensatz zu Befehlen werden Label mit dem Doppelpunkt abgeschlossen.

```
1 goto LABEL;
2 ...
3 LABEL:
```

Mithilfe von *goto* und Labels können Unterprogramme realisiert werden, um Code mehrfach zu verwenden. Es gibt in BusyC jedoch keinen *return*-Befehl, daher muss nach der Ausführung des Unterprogramms mit *goto* zurückgesprungen werden.

### Halten

Mit dem Befehl *halt* kann ein BusyC-Programm sofort zum Halten gebracht werden. Programme halten außerdem, sobald das Ende erreicht ist.

```
halt;
```

### Konditionale

Mit der *if* -Bedingung lassen sich Befehle bedingt ausführen. *if* -Bedingungen beziehen sich stets auf genau einen Befehl: Inkrement, Dekrement, Sprung oder den Haltebefehl. Bedingte Variablendefinitionen oder geschachtelte *if* -Bedingungen sind nicht erlaubt. Die möglichen Bedingungen sind Vergleiche von einer Variable und dem Wert 0. Direkt Vergleiche von Variablen sind nicht möglich.

```
if (var == 0) var++;
if (var != 0) var--;
```

Die Bedingung muss der Reihenfolge Variablenbezeichner, Vergleichsoperator, 0 folgen und geklammert sein.

### Kommentare

BusyC verwendet einen einzelnen Schrägstrich „/“ als Kommentarsymbol. Der Grund nicht den doppelten Schrägstrich wie in C dazu zu verlangen ist, dass ein einzelner Schrägstrich kein erlaubtes Zeichen in BusyC ist.

```
/ Dies ist ein Kommentar
```

### Beispiel

Der folgende Codeabschnitt zeigt ein das Programm *isEven.bc*, das genau dann anhält, wenn der initiale Wert von *b* eine gerade Zahl ist. Dafür wird *b* in einer Schleife dekrementiert und die Gleichheit mit 0 überprüft. Falls *b* nach einer geraden Anzahl an Dekrements den Wert 0 hat, war *b* gerade. Andernfalls war *b* ungerade.

```
uint b = 10;

CHECK:
if (b == 0) halt;
b--;
if (b == 0) goto INFINITE_LOOP;
b--;
goto CHECK;

INFINITE_LOOP:
goto INFINITE_LOOP;
```

## 4.2 BusyC Programme testen

Zum Testen entwickelter BusyC-Programme wurde ein Python-Skript geschrieben, das ein BusyC-Programm in ein fast[6] äquivalentes C-Programm umwandelt. Die folgenden Dinge werden verändert:

- Das Programm wird in eine *main*-Funktion geschrieben
- Der Typ *uint* wird mit einem Makro definiert
- Variablendefinitionen werden an den Anfang der *main*-Funktion geschrieben
- Der Dekrement-Befehl wird durch eine Funktion ersetzt, die prüft, ob die Variable schon den Wert 0 hat, um Underflows zu verhindern
- Der Haltebefehl wird durch *return 0* ersetzt
- *stdio.h* aus der C-Standardbibliothek wird inkludiert, damit *printf* genutzt werden kann

Der folgende Codeabschnitt zeigt ein C-Programm, das durch die Umwandlung von *isEven.bc* entstanden ist.

```
1  #include <stdio.h>
2  #define uint unsigned int
3
4  uint decrement(uint a) {
5      if (a > 0) a--;
6      return a;
7  }
8
9  int main() {
10 uint b = 10;
11
12 CHECK:
13 if (b == 0) return 0;
14 b = decrement(b);
15 if (b == 0) goto INFINITE_LOOP;
16 b = decrement(b);
17 goto CHECK;
18
19 INFINITE_LOOP:
20 goto INFINITE_LOOP;
21
22 return 0;
23 }
```

Die erzeugten C-Programme wurden anschließend mithilfe eines Debuggers getestet. Dieses Testen dient ausschließlich dazu sicherzustellen, dass die Programmlogik korrekt ist.

[6] Der einzige Unterschied ist, dass BusyC kein *uint*-Limit hat

# 5

# Parsing von BusyC

Dieses Kapitel ist das erste von zwei Kapiteln über den entwickelten BusyC-Compiler. Der Kompilierungsprozess kann in zwei Phasen aufgeteilt werden: Das Parsing des Quellcodes und das Übersetzen in eine Turingmaschine. In diesem Kapitel wird die Funktionsweise des Parsers erläutert.

### Begriffserklärung

Für die Beschreibung des Parsers werden die Begriffe *Wort* und *Ausdruck* verwendet. Ein Wort bezeichnet eine zusammenhängende Zeichenfolge im Quellcode, die für sich genommen eine Bedeutung hat. Beispiele dafür sind Variablenbezeichner, Zahlen oder Operatoren. Ein Ausdruck ist eine Folge von Worten, die ein Verhalten des Programms beschreibt. Dazu zählen Befehle, Konditionale und Labels.

## 5.1 String-Trees

Die *String-Tree*-Datenstruktur ist *die* essentielle Datenstruktur des Parsers. Sie erlaubt es einem Wort eine Bedeutung zuzuordnen und diese Bedeutung in linearer Zeit zu finden.

### Repräsentation in C

```
1 typedef struct StringTree {
2     struct StringTree** leaves;
3     Type t;
4     int id;
5     int strLength;
6 } StringTree;
```

Dieses Struct modelliert den Knoten eines String-Trees. Ein Knoten enthält ein Array *leaves* mit Pointern auf Knoten, eine Variable *t* für den Typ eines Worts, eine Variable *id* für weitere Spezifikation der Bedeutung des Worts und eine Variable *strLength* für die Länge des Worts.

Ein String-Tree fängt an der Spitze mit einem einzigen Knoten an. Kanten auf weitere Knoten können im Array *leaves* sein. Das Array hat immer die Länge 67 und kann damit für jedes erlaubte Zeichen[7] in BusyC einen Pointer auf einen weiteren Knoten beinhalten. Für das finden einer Kante im Array wird eine Lookup-Tabelle verwendet, die jedem erlaubten Zeichen einen Wert zwischen 0 und 66 zuordnet. Für Zeichen, für die es keine Kante gibt, liegt der Wert *NULL* an der dazugehörigen Stelle im Array.

Die Knoten und *leaves*-Arrays werden jeweils in separaten Arrays gespeichert, die alle Knoten bzw. *leaves*-Arrays beinhalten. Das *leaves*-Arrays-Array wird bei der Initialisierung genullt.

```
1 StringTree* tree = NULL;
2 StringTree** leafArrays = NULL;
3 ...
4 void initTree() {
5     // Allokation für 100 Knoten
6     tree = malloc(cap * sizeof(StringTree));
7     leafArrays = calloc(cap * NUMCHARS, sizeof(StringTree*)); // Wird
          genullt
8     ...
9 }
```

Abbildung 5.1 zeigt einen kleinen String-Tree mit 4 erlaubten Zeichen, der die Worte „bb“, „bus“, „busy“, „buy“, „by“, „sub“, „sus“ und „sys“ enthält. Die schwarzen Buchstaben repräsentieren Knoten, die ein Wortende darstellen.

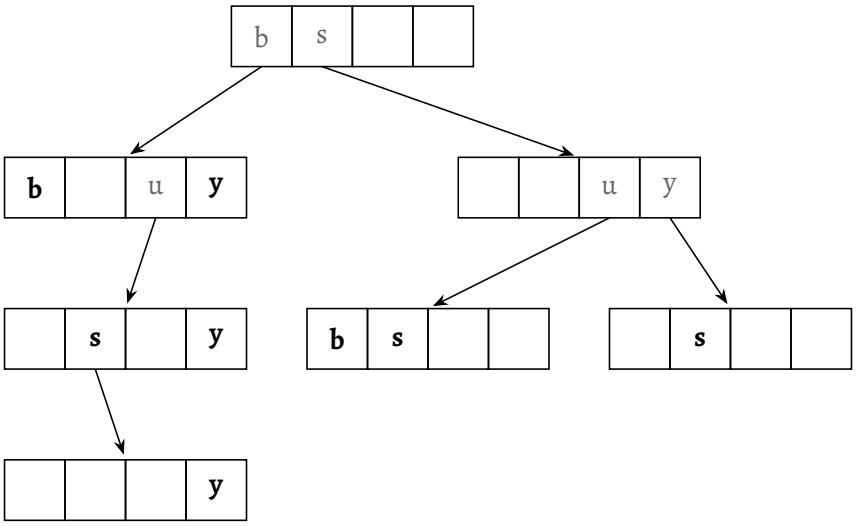



Abbildung 5.1: Ein String-Tree mit den Zeichen „b“, „s“, „u“ und „y“.

Für das Suchen eines Wortes wird über die Knoten iteriert, bis zum Ende des Wortes oder bis es keine Kante mehr gibt. Falls es keine Kante mehr gibt, befindet sich das Wort

[7] Abgesehen von den Zeichen für Klammern „(“ und „)“, da diese nicht in Worten vorkommen dürfen.

nicht im String-Tree. Die nächste Kante findet man im *leaves*-Array an der Stelle, die in der Lookup-Tabelle für das aktuelle Zeichen eingetragen ist.

Die Variable *t* vom Typ *Type* enthält den Typ eines Wortes. *Type* ist ein Enum und der folgende Codeabschnitt zeigt die Definition.

```
typedef enum {
    NOTHING_T = 0,
    VARIABLE_T,
    LABEL_T,
    INSTRUCTION_T,
    DECLARATION_T,
    CONDITIONAL_T,
    OPERATOR_T,
} Type;
```

Für Knoten, die kein Wortende sind, hat die Variable *t* den Wert *NOTHING_T*. Die Variable *id* beschreibt die Bedeutung eines Wertes genauer im Bezug zum Typ. Für Typ *INSTRUCTION_T* spezifiziert die *id* die Instruktion, für den Typ *VARIABLE_T* hat jede Variable eine eigene *id* für die Identifikation. Für andere Typen funktioniert das analog.

## 5.2 Parsen eines BusyC-Programms

Der Quellcode des BusyC-Programms liegt vor dem Kompilieren in einer separaten Datei vor. Die Datei wird Zeilenweise eingelesen, wobei das Lesen einer Zeile nach dem „/“–Symbol endet, um Kommentare zu entfernen. Das Programm wird Zeilenweise in einem 2D-Array gespeichert. Jede Zeile wird anschließend in Ausdrücke getrennt anhand des Semikolons bzw. Doppelpunkts. Dabei wird beachtet, dass ein Doppelpunkt jedes mal und nur nach einem Label kommt. Die Ausdrücke werden dabei nacheinander in ein 1D-Array geschrieben[8].

Die isolierten Ausdrücke werden Wort für Wort gelesen, wobei den Worten mithilfe der String-Tree Datenstruktur ihre Bedeutung zugeordnet wird. Das folgende Beispiel demonstriert die Vorgehensweise des Parsers beim parsen eines Ausdrucks.

```
uint beaver5 = 47176870
```

1. Suche das erste Wort im String-Tree
2. *uint* gefunden, daher ist dieser Ausdruck eine Variablendefinitionen
3. Prüfe, dass das nächste Wort ein unbenutzter Variablenbezeichner ist durch Suche im String-Tree
4. Schreibe den unbenutzten Variablenbezeichner in den String-Tree
5. Prüfe, dass das nächste Wort der „=“-Operator ist
6. Prüfe, dass das nächste Wort nur aus Ziffern besteht und schreibe es in einen Puffer
7. Prüfe, dass kein Wort mehr folgt und iteriere bis zum Nullbyte

[8] Ein 1D-Array aus *char*-Pointern bzw. Strings.

Alle anderen Arten von Ausdrücken werden analog geparst, mit dem Unterschied welche Worte erwartet werden. Die Art des Ausdrucks kann für alle Ausrücke am ersten Wort erkannt werden.

## 5.3 Interne Repräsentation eines Ausdrucks

Der Zweck des Parsings ist das Programm in eine interne Repräsentation zu übersetzen, um dieses später in eine Turingmaschine zu übersetzen. Für die Repräsentation der Ausdrücke wird ein Struct verwendet, welches aus einem Enum und einer Union besteht, welches drei Structs vereint …

Für das Verständnis des Parsers ist die genaue Zusammensetzung der Datenstruktur mit allen Verschachtelungen nicht notwendig. Der folgende Codeabschnitt zeigt sie trotzdem, jedoch ist es nicht wichtig diese im Detail zu verstehen. Wichtig zu verstehen ist nur, dass das Struct *ExpressionValue* einen Ausdruck als Union speichert und einen Enum-Typ nutzt, um die Union richtig zu interpretieren.

```
typedef enum {
    DECLARATION,
    CONDITIONAL,
    INSTRUCTION,
    LABEL
} ExpressionType;

typedef enum {
    INCREMENT,
    DECREMENT,
    GOTO,
    HALT,
    IF_ZERO,
    IF_NOT_ZERO
} InstructionType;

typedef union {
    struct {
        int numVar;
        int initVal;
    } declaration;
    struct {
        InstructionType iType;
        int address; // Adresse des Befehls
        union { // union !!!
            int varID;
            int jumpAdress; // Sprungadresse für goto
            char* labelName; // Labelname bevor Adresse daraus berechnet
                wird
        } param;
    } instruction;
    struct {
```

```
32          int address; // Adresse des ersten Befehls nach Label
33      } label;
34  } ExpressionValue;
35
36  typedef struct {
37      ExpressionType eType;
38      ExpressionValue eVal;
39  } Expression;
```

Das gesamte Programm liegt nach dem Parsen in einem *Expression*-Array vor. Außerdem gibt es vier weitere Arrays, die jeweils Pointer auf die Variablendefinitionen, Befehle, Konditionale und Labels speichern. Verwendet werden davon nur das Array für Befehle und Labels, um auf den $i$-ten Befehl / das $j$-te Label zuzugreifen.

## 5.4 Sprünge zu ungeparsten Labels

Im Gegensatz zu Variablen können Labels schon vor ihrer Definition verwendet werden, nämlich um „von oben“ zu ihnen zu springen. Beim Parsing eines Sprungs ist das insofern ein Problem, dass für ungeparste Labels noch nicht klar ist wohin gesprungen werden muss.

Der Parser löst das, indem während des Parsings für jedes *goto* zunächst ein Pointer auf den Label-String gespeichert wird. Nachdem das gesamte Programm geparst wurde, wird über alle *gotos* iteriert[9] und mithilfe des String-Trees und dem Label-Array die Sprungadresse ermittelt.

[9] Eigentlich wird über das Befehle-Array iteriert, wobei für alle anderen Befehle nichts passiert.

# 6

# Übersetzung in eine Maschine

Im zweiten Kapitel über den entwickelten BusyC-Compiler wird die Übersetzung des Quellcodes in eine Turingmaschine behandelt. Es werden die Submaschinen vorgestellt, die die Befehle ausführen und erklärt wie diese Submaschinen zu einer großen Maschine zusammengebaut werden, die das, durch den Quellcode spezifizierte Programm, ausführt.

Um in diesem Kapitel Variablenwerte und Zellenwerte vom Maschinenband nicht zu verwechseln, werden Variablenwerte mit Ziffern dargestellt, während Zellenwerte als „Eins“ und „Null“ ausgeschrieben werden.

## 6.1 Submaschinen für Operationen auf Variablen

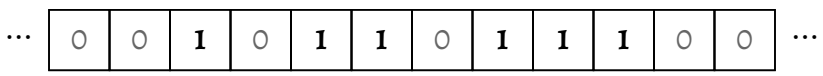
Definition 6.1 (Submaschine). Eine *Submaschine* ist eine Turingmaschine, die als Teil einer größeren Turingmaschine eine wohldefinierte Aufgabe ausführt.

Die Zustände von Submaschinen werden mit dem Buchstaben „s“ beschriftet.

### Variablen auf dem Maschinenband

Die Variablen des Programms liegen auf dem Maschinenband. Sie sind unär kodiert, wobei eine Zahl $n$ durch eine Folge von $n + 1$ Einsen auf dem Band repräsentiert wird, damit die Zahl 0 auch dargestellt werden kann. Variablen werden jeweils durch *exakt* eine Null auf dem Band getrennt.

Es folgt ein Beispiel, bei dem drei Variablen mit den Werten 0, 1, 2 auf dem Band liegen.



## Die Inkrement-Submaschine

Die Inkrement-Submaschine hat die Aufgabe eine Variable auf dem Band zu inkrementieren. Da die Variablen unär kodiert sind, wird zum Inkrementieren eine Eins an das rechte Ende der Variable hinzugefügt. Das Band ist so organisiert, dass zwischen Variablen exakt eine Null liegt. Um diese Form einzuhalten, werden nachfolgende Variablen um eine Zelle nach rechts verschoben.

Damit die Inkrement-Submaschine eine Variable inkrementieren kann, muss beim Start der Kopf auf einer beliebigen Eins stehen, die zur Variable gehört. Beim Einsatz in einer vollständigen Maschine startet der Kopf auf der ersten Eins (von links nach rechts) einer Variable.

Abbildung 6.1 zeigt die Inkrement-Submaschine. Danach folgt eine Simulation der Maschine auf einem Band mit drei Variablen, von denen die erste inkrementiert wird.

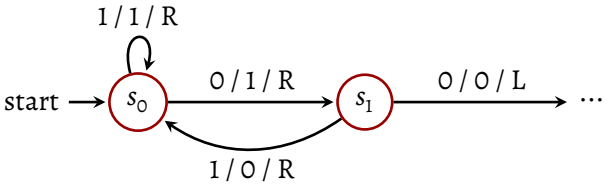
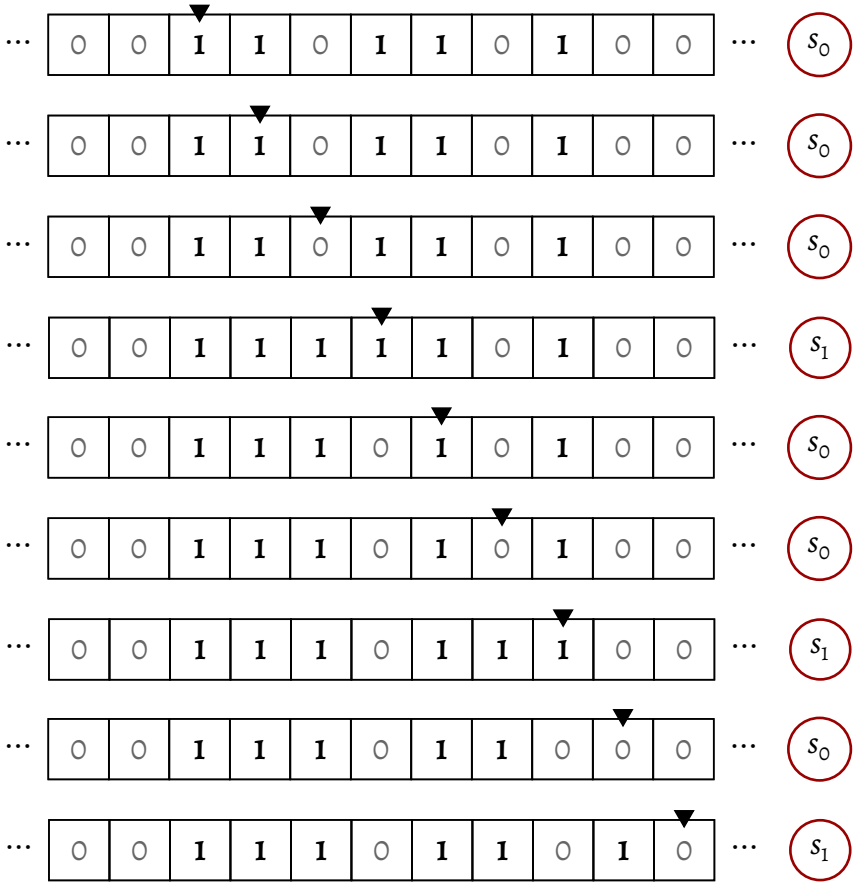



Abbildung 6.1: Die Inkrement-Submaschine



Die Simulation zeigt, wie die Inkrement-Submaschine arbeitet. Die zu inkrementierende Variable wird um eine Eins verlängert und nachfolgende Variablen werden um eine Zelle nach rechts verschoben. Zum Schluss wechselt die Maschine zur Rückkehr–Submaschine, die später in diesem Kapitel vorgestellt wird.

## Die Dekrement-Submaschine

Die Dekrement-Submaschine hat die Aufgabe eine Variable auf dem Band zu dekrementieren. Zum Dekrementieren einer Variable, muss diese um eine Eins gekürzt werden. Dabei werden die nachfolgenden Variablen um eine Zelle nach links verschoben, damit die Form des Bands eingehalten wird. Variablen sind vom Typ *uint*, also nicht-negative ganze Zahlen. Daher ist die Dekrement-Operation in BusyC so definiert, dass die Null auf sich selbst abgebildet wird. Um dieses Verhalten zu emulieren prüft die Dekrement-Submaschine dafür, ob die Variable den Wert 0 hat und wechselt in dem Fall zur Rückkehr-Submaschine.

Abbildung 6.1 zeigt die Dekrement-Submaschine. Danach folgt eine Simulation der Maschine auf einem Band mit einer Variable mit dem Wert 2.

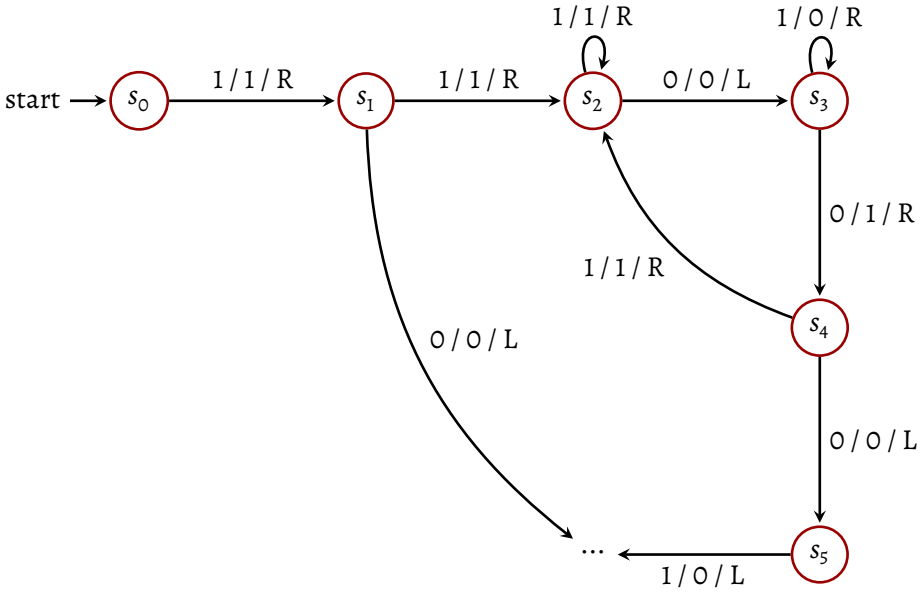
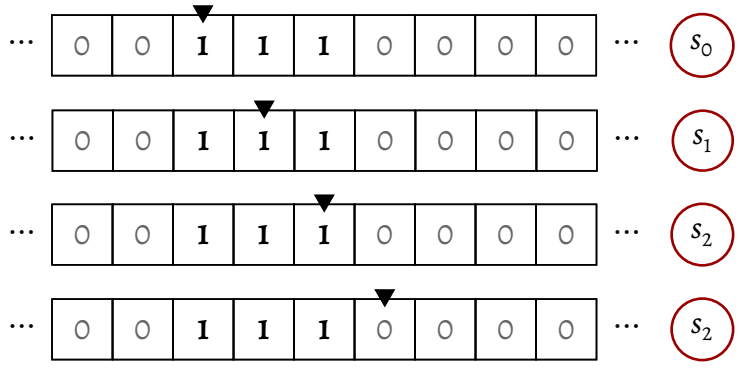
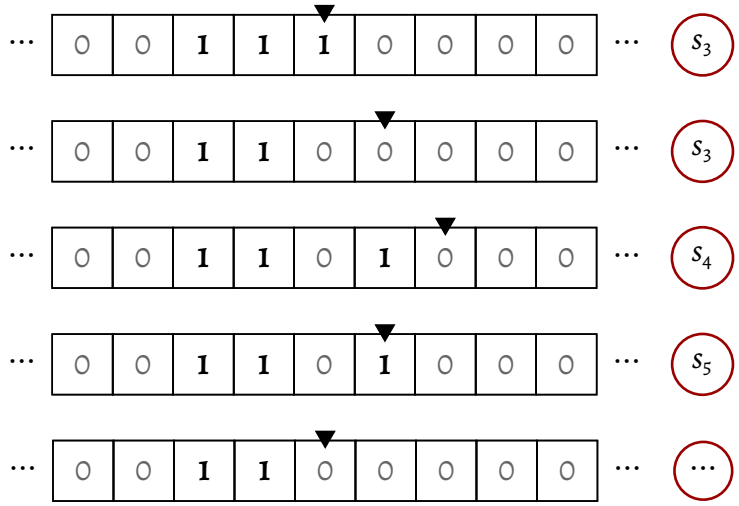



Abbildung 6.2: Die Dekrement-Submaschine






Hätte die Variable den Wert 0 gehabt, hätte die Maschine im zweiten Schritt der Simulation im Zustand $s_1$ die Berechnung abgebrochen und sofort zur Rückkehr-Submaschine gewechselt, wodurch das Band unverändert geblieben wäre.
Ausgelassene Kanten[10] bei dieser Maschine könnten beliebig gewählt werden und haben keinen Einfluss auf die Berechnungen der Dekrement-Submaschine.

### Submaschinen für Konditionale

Konditionale in BusyC erlauben die Bedingungen Gleichheit und Ungleichheit zu Null einer Variable. Für diese Bedingungen gibt es zwei Submaschinen, die gleich aufgebaut sind und sich nur darin Unterscheiden, zu welcher Submaschine nach dem Prüfen der Bedingung gewechselt werden soll. Falls die Bedingung wahr ist, wird zur Rückkehr–Submaschine gewechselt, andernfalls zur Rückkehr+-Submaschine.

Zum prüfen der Bedingung prüfen die Submaschinen die Zelle rechts neben der ersten Zelle einer Variable. Diese Zelle hat genau dann den Wert Null, wenn die Variable den Wert 0 hat. Abhängig davon entscheidet die jeweilige Submaschine, zu welcher Submaschine als nächstes gewechselt wird.

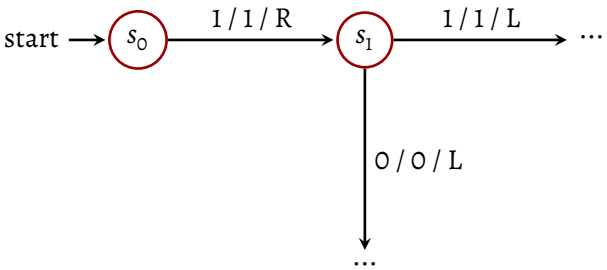



Abbildung 6.3: Eine Submaschine zum Vergleichen von einer Variable mit 0

[10] Zum Beispiel fehlt im Zustand $s_0$ eine Kante für den nächsten Schritt der Maschine, falls eine Null gelesen wird.

## 6.2 Das Steuerwerk der Maschine

Die Submaschinen des letzten Abschnitts bilden das „Operationswerk“ der Maschine. In diesem Abschnitt werden die Submaschinen vorgestellt, die das „Steuerwerk“ bilden.

### Der Programmzähler

Der Programmzähler ist eine weitere unär kodierte Zahl, die zusätzlich zu den Variablen auf dem Band steht. Der Programmzähler befindet sich links von den Variablen und ist durch *exakt* eine Null von der ersten Variable getrennt.

Der Programmzähler speichert welcher Befehl des Programms als nächstes ausgeführt werden muss. Für die Ansteuerung eines Befehls mithilfe des Programmzählers gibt es die Befehlsauswahl-Submaschine, die von links nach rechts über den Programmzähler iteriert und zu einem Befehl wechselt, sobald die trennende Null erreicht ist. Im Gegensatz zu den bisherigen Submaschinen, die statisch per Hand vor dem Kompilieren erstellt wurden, wird diese Submaschine dynamisch während der Kompilierung erstellt, da sie je nach Programm unterschiedlich ist.

Es folgt ein Beispielprogramm in BusyC und die dazugehörige Befehlsauswahl–Submaschine in Abbildung 6.2.

```
1 uint a = 0;
2 uint b = 0;
3 uint c = 0;
4 a++;
5 b++;
6 c++;
```

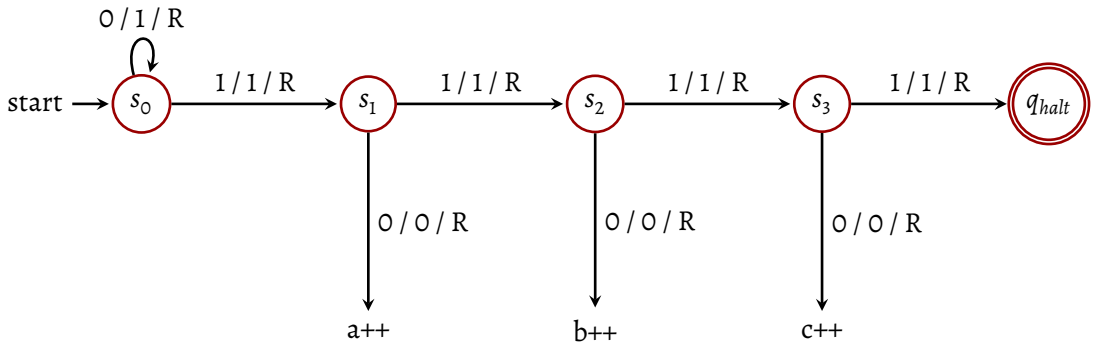
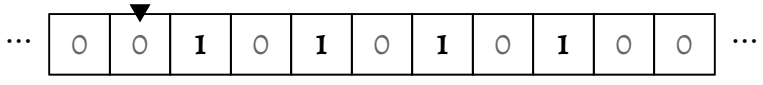



Abbildung 6.4: Eine Befehlsauswahl-Submaschine

Die Befehlsauswahl-Submaschine startet mit dem Kopf auf der Zelle unmittelbar links von dem Programmzähler im Zustand $s_0$. Die folgende Abbildung zeigt das Band der Maschine zu dem Beispielprogramm, bevor der erste Befehl ausgeführt wurde.



Die Befehlsauswahl-Submaschine erhöht den Programmzähler, indem im Zustand $s_0$ die Null zu einer Eins überschrieben wird. Danach zählt die Maschine die nachfolgenden Einsen und iteriert dabei über ihre Zustände. Sobald die trennende Null erreicht wird, wechselt der Zustand zu einer Submaschine, die den Befehl ausführt. In diesem konkreten Beispiel enthält der Programmzähler nur eine Eins, weshalb die Befehlsauswahl–Submaschine im Zustand $s_1$ die Null liest und zur Submaschine für den Befehl *a++* wechselt. Da der Programmzähler um eine Eins gewachsen ist, wird der Befehlsauswahl–Submaschine in der nächsten Ausführung bis zum Zustand $s_2$ kommen und zu Befehl *b++* wechseln. Auf diese Weise werden nach und nach alle Befehle des Programms ausgeführt.

Am Ende des Programms, nach der Ausführung des letzten Befehls, ist der Programmzähler so groß, dass die gesamte Befehlsauswahl-Submaschine durchlaufen wird und der Haltezustand erreicht wird.

## Die Rückkehr-Submaschine

Die Rückkehr-Submaschine hat die Aufgabe nach der Ausführung eines Befehls den Kopf der Maschine wieder auf die Position zu bringen, auf der die Befehlsauswahl-Submaschine den nächsten Befehl ansteuern kann, sodass sich eine Ausführungsschleife ergibt, die das gesamte Programm ausführt.

Abbildung 6.2 zeigt die Rückkehr-Submaschine. Danach folgt eine Simulation der Maschine. Die Simulation startet, nachdem die Variable auf dem Band im ersten Befehl inkrementiert wurde.

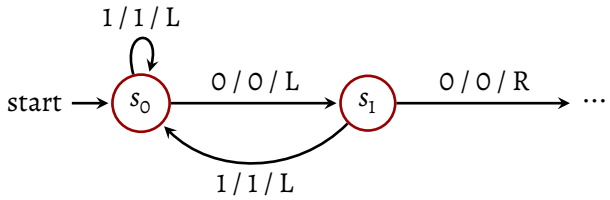
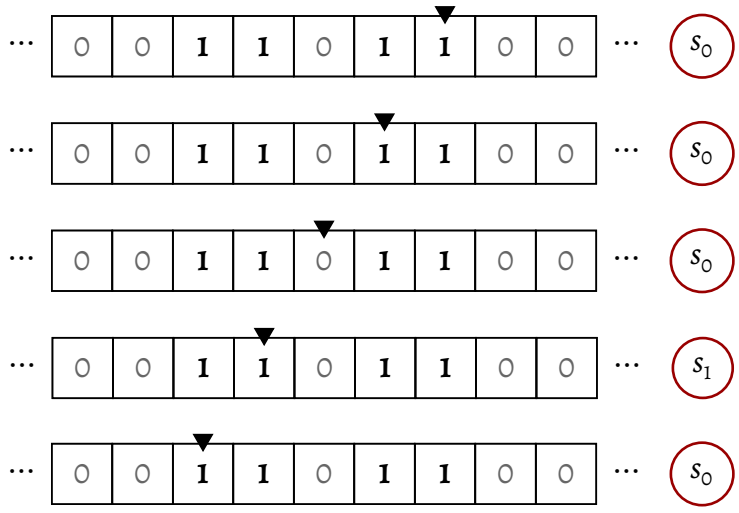
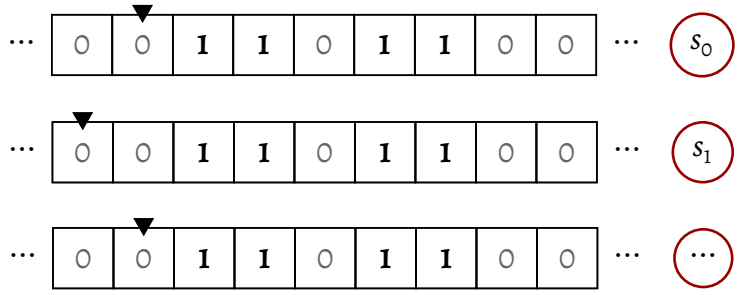



Abbildung 6.5: Die Rückkehr-Submaschine





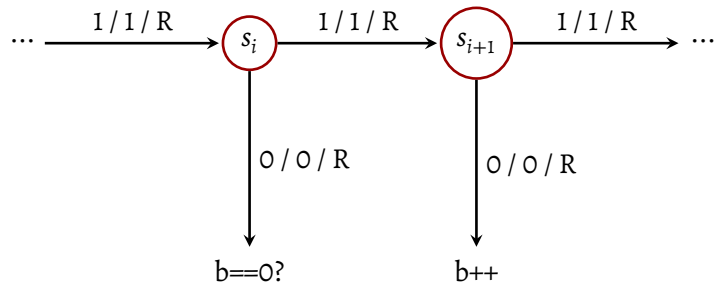

Die Rückkehr-Submaschine lässt den Kopf bis an das linke Ende des Programmzählers laufen und positioniert ihn auf die Zelle links vom Programmzähler und wechselt den Zustand zur Befehlsauswahl-Submaschine. Diese Steuert dann den nächsten Befehl an. Da die Befehlsauswahl-Submaschine den Programmzähler bei jedem Durchlauf inkrementiert, wird beim nächsten Mal der nächste Befehl angesteuert. Nach dem Befehl kehrt der Kopf mit der Rückkehr-Submaschine zurück …Diese Konstruktion bildet eine Schleife, in der das gesamte Programm ausgeführt wird.

## Befehle bedingt ausführen

Im Abschnitt *Submaschinen für Operationen auf Variablen* wurden Submaschinen eingeführt, die prüfen, ob eine Variable den Wert 0 hat. Nach dem Prüfen dieser Bedingung wechselt die Maschine entweder zur Rückkehr- oder Rückkehr+-Submaschine, abhängig davon, ob die Bedingung erfüllt wurde. Ein Konditional wie

```
1 if (b == 0) b++;
```

wird in zwei Befehle aufgeteilt: Die Bedingungsprüfung und den Befehl, der bedingt ausgeführt werden soll. Abbildung 6.2 zeigt, wie das in der Befehlsauswahl-Submaschine aussieht.




Falls die Bedingung als korrekt ausgewertet wird, wechselt die prüfende Submaschine zu der Rückkehr-Submaschine. Das führt dazu, dass der bedingte Befehl normal ausgeführt wird.

Wird die Bedingung als falsch ausgewertet, wechselt die prüfende Submaschine zu der Rückkehr+-Submaschine. Diese Submaschine funktioniert ähnlich wie die Rückkehr–Submaschine, mit dem Unterschied, dass der Programmzähler vor dem Wechsel zu der Befehlsauswahl-Submaschine inkrementiert wird, was das Überspringen des bedingten

Befehls bewirkt. Abbildung 6.2 zeigt die Submaschine, danach folgt eine Simulation der Maschine.

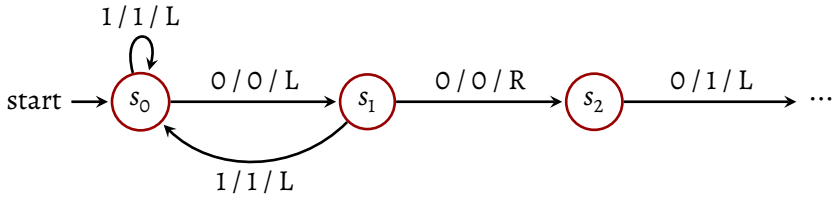
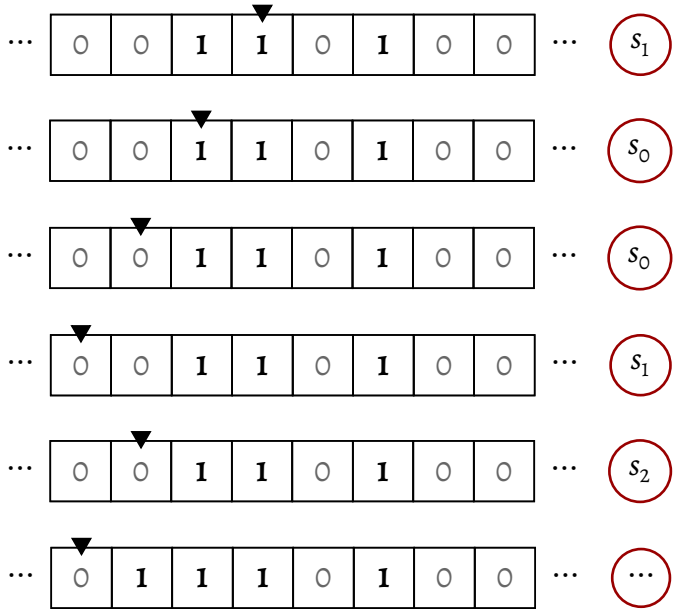



Abbildung 6.6: Die Rückkehr+-Submaschine




Die Simulation zeigt das Inkrementieren des Programmzählers vor dem Wechsel auf die Befehlsauswahl-Submaschine. Dies bewirkt das Überspringen des nächsten Befehls.

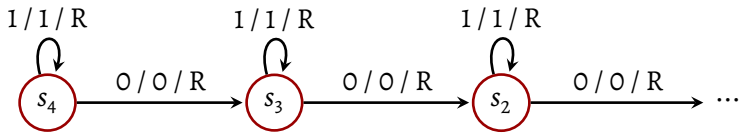
## Adressierung von Variablen

Submaschinen, die auf Variablen arbeiten[11]. setzen voraus, dass sich der Kopf der Maschine beim Start auf der ersten Zelle der Variable befindet. Das für die erste Variable auf dem Band, denn nach dem Wechsel Befehlsauswahl-Submaschine zur Submaschine für die Ausführung des Befehls steht der Kopf der Maschine auf genau dieser Zelle. Um auch mit den weiteren Variablen rechnen zu können, wird eine Submaschine benötigt, die die weiteren Variablen auf dem Band adressieren kann.

Die entsprechende Submaschine ist die Adressierung-Submaschine, welche den Kopf der Maschine auf die erste Zelle einer beliebigen Variable bewegt. Dazu nutzt sie die Form des Bandes, indem die trennenden Nullen „gezählt“ werden. Da die Anzahl der Variablen vom Programm abhängt wird diese Submaschine dynamisch während der Kompilierung

[11] Inkrement-, Dekrement- und Submaschinen für den Vergleich mit 0.

erstellt. Abbildung 6.2 zeigt eine Adressierung-Submaschine für ein Programm mit vier Variablen.




Diese Adressierung-Submaschine hat drei Zustände, obwohl sie für die Adressierung von vier Variablen zuständig ist. Außerdem hat sie, im Gegensatz zu allen vorherigen Submaschinen, keinen festgelegten *Startzustand*.

Dieser hängt nämlich davon ab, welche Variable adressiert werden soll. Zum adressieren der ersten Variable, wechselt der Zustand direkt zu der Submaschine, die den Befehl ausführt, da sich der Kopf nach der Ausführung der Befehlsauswahl-Submaschine bereits auf der ersten Variable befindet.

Die Zustände in Abbildung 6.2 wurden so benannt, dass für das Adressieren der $i$-ten Variable in den Zustand $s_i$ gewechselt werden soll. Die Adressierung-Submaschine funktioniert so, dass jeder Zustand der Sequenz eine Variable überspringt. Der letzte Zustand der Sequenz adressiert die zweite Variable, der vorletzte Zustand der Sequenz adressiert die dritte Variable …

Nach dem letzten Zustand der Sequenz wechselt der Zustand zu der Submaschine, die den Befehl ausführt. Damit jeder Befehl für alle Variablen ausgeführt werden kann, benötigt jede befehlsausführende Submaschine eine eigene Adressierung-Submaschine. Diese sind alle gleich aufgebaut und unterscheiden sich nur darin, zu welcher Submaschine am Ende der Sequenz gewechselt wird.

Enthält das Programm beispielsweise den Befehl die zweite Variable zu inkrementieren, hat die Befehlsauswahl-Submaschine einen Zustand, der diesen Befehl ansteuert. Dabei wird zu dem Zustand in der Adressierung-Submaschine für den Inkrement-Befehl gewechselt, der die zweite Variable adressiert.

Insgesamt benötigt eine Maschine vier Adressierung-Submaschinen für diese Operationen:

- Inkrement
- Dekrement
- Prüfen ob Variable == 0
- Prüfen ob Variable != 0

## 6.3 Umsetzung von Sprungbefehlen

Im letzten Abschnitt wurde erläutert, wie kompilierte Maschinen mithilfe des Programmzählers Befehl für Befehl Programme ausführen. In diesem Abschnitt werden Submaschinen eingeführt, die Sprungbefehle ermöglichen, indem sie den Wert des Programmzählers verändern. Eine Besonderheit von Sprungbefehlen ist, dass sich der Kopf der Maschine für einen Sprungbefehl nach links auf den Programmzähler bewegt, wenn der Zustand von

der Befehlsauswahl-Submaschine zum Sprungbefehl ändert. Bei allen anderen Befehlen bewegt sich der Kopf nach rechts zu den Variablen. Abbildung 6.3 zeigt einen Abschnitt aus einer Befehlsauswahl-Submaschine mit einem Sprungbefehl und einem beliebigen anderen Befehl. Man beachte die Schieberichtung des Kopfes.

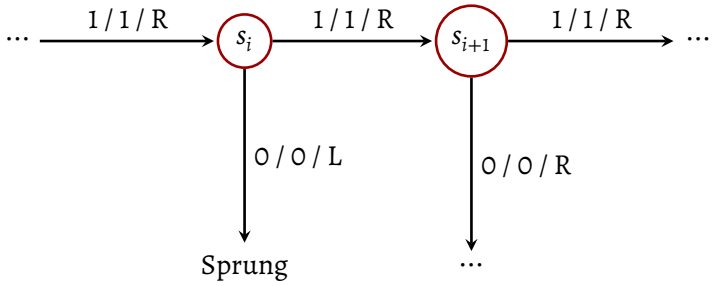



Abbildung 6.7: Eine Befehlsauswahl-Submaschine, die einen Sprungbefehl enthält

Interpretiert man den Programmzähler als unär kodierte Zahl wie die Variablen, kann jedem Befehl im Programm ein Wert des Programmzählers zugeordnet werden, bei dem dieser Befehl angesteuert wird. Für die Realisierung eines Sprungbefehls muss die Differenz zwischen dem aktuellen Wert des Programmzählers und des Wertes für den Zielbefehl auf den Programmzähler addiert bzw. von ihm subtrahiert werden.

### Positive Sprünge

Ein positiver Sprung ist ein Sprung zu einer späteren Stelle im Programm. Das bedeutet für die Umsetzung, dass der Wert des Programmzählers erhöht werden muss. Schauen wir uns zunächst den Sonderfall an, dass zum unmittelbar nächsten Befehl gesprungen wird.

```
1 goto ZIEL;
2 ZIEL:
3 befehl;
```

In diesem Fall muss der Programmzähler nicht verändert werden. Die Befehlsauswahl–Submaschine steuert einfach die Rückkehr-Submaschine an.

Für Sprünge, bei denen mindestens ein Befehl übersprungen wird, wird eine Submaschine genutzt. Diese ist ähnlich zu den Submaschinen für die Adressierung von Variablen in der Hinsicht, dass sie keinen festen Startzustand hat und je nach Sprung von einem anderen Zustand gestartet wird. Diese Submaschine ist eine Sequenz von Zuständen, die den Programmzähler um jeweils eine Eins erweitern und danach zum nächsten Zustand in der Sequenz wechseln. Abbildung 6.3 zeigt eine beispielhaften Sequenz von vier Zuständen für positive Sprünge.

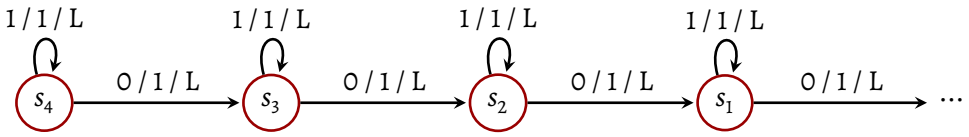
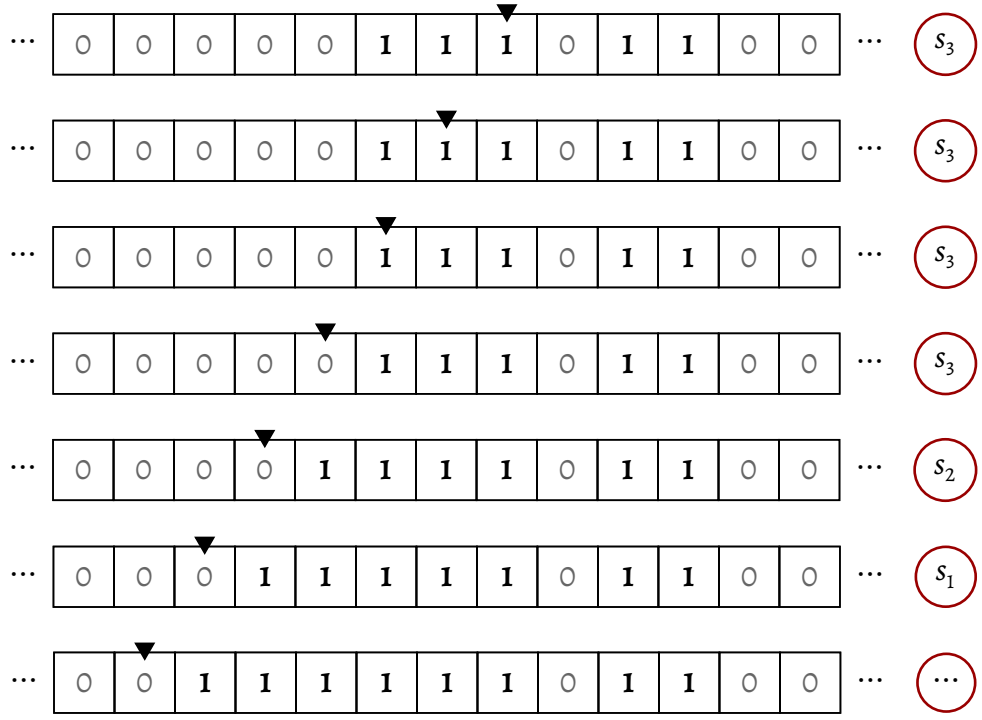



Abbildung 6.8: Eine Sequenz für positive Sprünge

Um den Programmzähler um $i$ zu erhöhen, muss der Zustand $s_i$ angesteuert werden. Die Submaschine schreibt weitere Einsen an das linke Ende des Programmzählers. Für das positionieren des Kopfes an das linke Ende des Programmzählers haben alle Zustände der Sequenz eine Kante die Eigenschaft beim lesen einer Eins den Kopf nach links zu bewegen und im Zustand zu bleiben. Es folgt eine Simulation eines Sprungs, der drei Befehle überspringt.




Am Ende wird in den ersten Zustand der Befehlsauswahl-Submaschine gewechselt, der die Zelle, auf der der Kopf steht zu einer Eins überschreibt. Danach wird der angesprungene Befehl angesteuert.

### Negative Sprünge

Ein negativer Sprung ist ein Sprung zu einem Befehl, der vorher im Programm auftaucht. Der Sonderfall, dass zu dem Sprungbefehl selbst gesprungen wird ist ein negativer Sprung.

```
1 LOOP:
2 goto LOOP;
```

Im Gegensatz zu positiven Sprüngen, bei denen der Programmzähler erhöht wird, subtrahieren negative Sprünge den Programmzähler. Dafür wird auch eine Sequenz von Zuständen genutzt, die Einsen zu Nullen überschreibt. Anders als bei positiven Sprüngen benötigen negative Sprünge einen eigenen Zustand, um den Kopf auf das linke Ende des Programmzählers zu bewegen. Bei positiven Sprüngen war dies nicht nötig, da die Zustände der Sequenz selbst den Kopf an das linke Ende des Programmzählers bewegt haben, da ihre Hauptaufgabe das Überschreiben von Nullen zu Einsen waren und sie beim Lesen von Einsen diese stehen lassen können. Die Zustände in der Sequenz der negativen Sprünge überschreiben Einsen zu Nullen, weshalb ein weiterer Zustand benötigt wird, der den Kopf über die Einsen bewegt. Wir nennen diesen Zustand *Adressierungszustand*. Negative Sprünge mit unterschiedlichen Reichweiten benötigen jeweils ihren eigenen Adressierungszustand. Negative Sprünge mit derselben Sprungweite nutzen denselben Adressierungszustand. Abbildung 6.3 zeigt eine Sequenz für negative Sprünge mit einem Adressierungszustand $a_0$. Nach der Abbildung folgt eine Simulation eines negativen Sprungs.

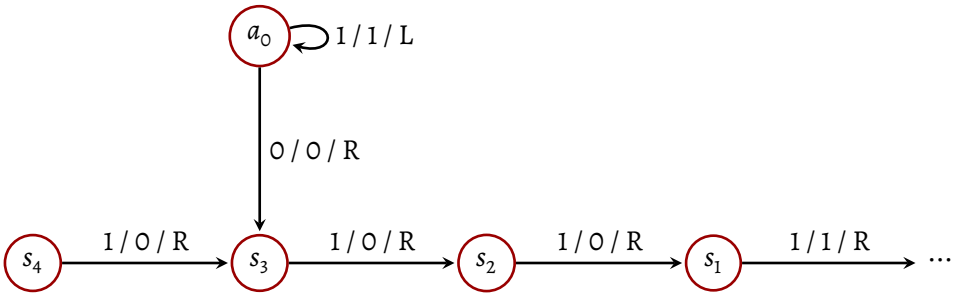
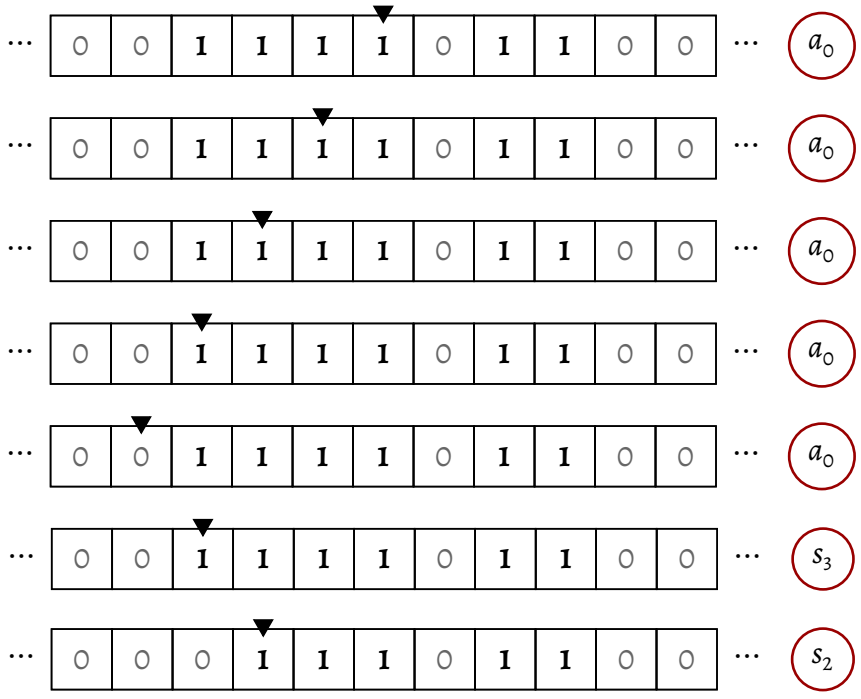
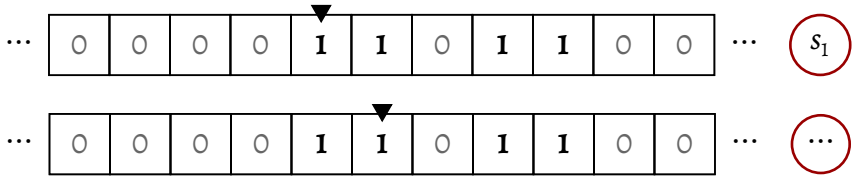



Abbildung 6.9: Eine Sequenz für negative Sprünge






In dieser Simulation wurde vom dritten Befehl zum ersten Befehl gesprungen. Die Befehlsauswahl-Submaschine hatte den Programmzähler schon inkrementiert, um in der nächsten Ausführung den vierten Befehl anzusteuern. Für den Sprung wurde der Programmzähler um zwei gekürzt. Eigentlich würde eine Kürzung von zwei einen Sprung von zwei Befehlen nach hinten bewirken, das Programm springt trotzdem drei Befehle zurück. Das liegt daran, auf welcher Zelle zur Befehlsauswahl-Submaschine gewechselt wird. In der Simulation ist erkennbar, dass der Zustand $s_1$ keine Eins vom Programmzähler entfernt. Stattdessen bewegt dieser den Kopf nur nach rechts, lässt die Eins stehen und wechselt zur Befehlsauswahl-Submaschine. Das hat dieselbe Wirkung, als wäre die Eins entfernt worden und danach durch die Befehlsauswahl-Submaschine wieder geschrieben worden.

## 6.4 Initialisierung des Bandes

Turingmaschinen starten mit einem genullten Band. Unsere kompilierten Maschinen speichern den Programmzähler und Variablen auf dem Band. Für das Schreiben des Programmzählers und der Variablen auf das Band wird eine Submaschine genutzt, die dynamisch während der Kompilierung erzeugt wird. Diese schreibt den Programmzähler initial als eine Eins auf das Band (also mit dem Wert 0) und alle Variablen mit genau dem Wert, den sie bei ihrer Definition zugewiesen bekommen haben. Die Variablenreihenfolge entspricht der Deklarationsreihenfolge. Zwischen Programmzähler und der ersten Variable und zwischen den Variablen wird immer jeweils genau eine trennende Null gelassen. Abbildung 6.4 zeigt die Initialisierung-Submaschine zu dem folgenden Codeabschnitt. Anschließend folgt eine Simulation der Maschine.

```
1 uint a = 0;
2 uint b = 1;
```

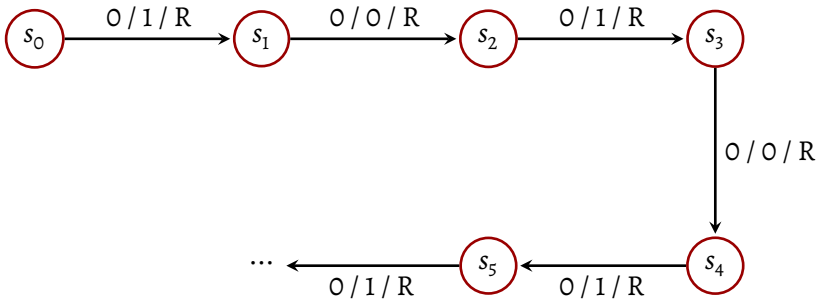
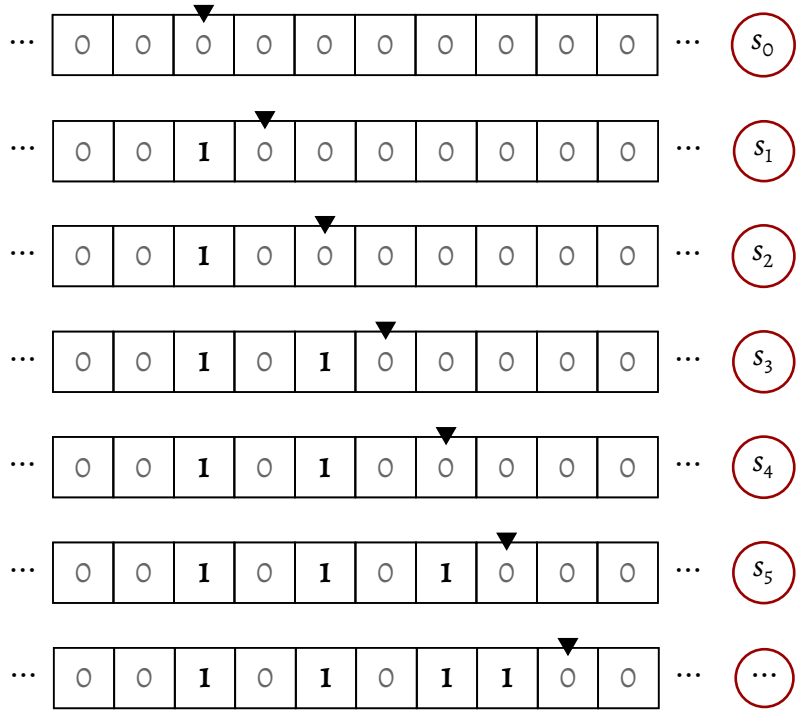



Abbildung 6.10: Eine Initialisierung-Submaschine




Nach der Initialisierung wechselt der Zustand zu der Rückkehr-Submaschine, die die Ausführungsschleife des Programms startet. Es wäre auch möglich gewesen das Band von rechts nach links zu initialisieren und direkt zur Befehlsauswahl-Submaschine zu wechseln. Das macht für die Anzahl der Zustände keinen Unterschied, da die Rückkehr–Submaschine sowieso in der Maschine enthalten ist[12]. Die Initialisierung von links nach rechts war simpler zu implementieren, da der erste Zustand der Maschine auf diese Weise immer den Programmzähler auf das Band schreibt und dieser daher schon am Anfang der Kompilierung in den Speicher geschrieben werden kann.

## 6.5 Die Größe der kompilierten Maschine

In diesem Abschnitt wird eine Worst-Case-Abschätzung für die Größe der erzeugten Maschine berechnet. Es werden die Konstanten $B$ für die Anzahl der Befehle, $V$ für die Anzahl der Variablen, $S$ für die Summe der Initialwerte der Variablen und $N$ für die Anzahl der negativen Sprunggrößen genutzt. Die Größe der generierten Maschine ergibt sich aus der Summe der Größen aller Submaschinen. Die statischen Submaschinen haben die Größen

- Inkrement-Submaschine: 2
- Dekrement-Submaschine: 6
- Variable-Gleich-Null-Submaschine: 2
- Variable-Ungleich-Null-Submaschine: 2

[12] Bei Programmen ohne Befehle wäre die Rückkehr-Submaschine tatsächlich nicht enthalten, wenn das Band auf diese Weise initialisiert werden würde. Programme ohne Befehle zu kompilieren wäre aber sinnlos. Mehr zu dem fehlen"von Submaschinen gibt es im nächsten Abschnitt

- Rückkehr-Submaschine: 2
- Rückkehr+-Submaschine: 3

Gesamt: 17

Die dynamischen Submaschinen haben im Worst-Case die Größen

- Befehlsauswahl-Submaschine: $B + 1$
- Adressierung-Submaschinen: $4 \cdot (V - 1)$
- Positive Goto-Sequenz: $B - 1$
- Negative Goto-Sequenz: $B + 1 + N$
- Initialisierung-Sequenz: $1 + 2V + S$

Gesamt: $3B + 6V + S + N - 2$

Zusammengerechnet ergibt das eine Größe von $3B + 6V + S + N + 15$ im Worst-Case. Die Größe der Maschine ist in $O(B + V + S + N)$, also linear.

## Optimierungen

Beim Kompilieren werden zunächst naiv alle statischen Submaschinen, die vier vollständigen Adressierung-Submaschinen und Adressierungszustände für jeden Zustand der negativen Sprungsequenz in die Gesamtmaschine eingebaut. Am Ende der Kompilierung werden die nicht erreichbaren Zustände der Gesamtmaschine mit Breitensuche herausgefiltert und entfernt. In der Praxis entfallen dadurch viele Zustände. Es entfallen ganze Submaschinen, wie z.B. die Dekrement-Submaschine und die zugehörige Adressierung–Submaschine, falls das Programm den entsprechenden Befehl nicht enthält.

Weiterhin können Programme optimiert werden, durch das ändern der Variablenreihenfolge und das aufteilen von Sprungbefehlen. Diese Optimierungen werden nicht vom Compiler übernommen und müssen manuell vom Programmierer vorgenommen werden. Es folgt ein Beispiel, welches das ändern der Variablenreihenfolge als Optimierung demonstriert.

```
1 uint a = 0;
2 uint b = 0;
3 uint c = 0;
4 if (c == 0) c++;
5 if (c != 0) c--;
```

Kompiliert man dieses Programm erhält man eine Maschine mit 37 Zuständen. Ändert man die Reihenfolge der Variablen und deklariert $c$ als erstes, entfallen die Adressierung–Submaschinen.

```
1 uint c = 0;
2 uint a = 0;
3 uint b = 0;
4 if (c == 0) c++;
5 if (c != 0) c--;
```

Dieses Programm wird zu einer Maschine mit 29 Zuständen kompiliert, da die vier Adressierung-Submaschinen wegfallen. Wäre *c* als zweites deklariert worden, hätte die Maschine 33 Zustände.

Das nächste Beispiel demonstriert das Aufteilen von Sprüngen als Optimierung.

```
goto END;
halt; halt; halt; halt; halt;
halt; halt; halt; halt; halt;
END:
```

Dieses Programm wird zu einer Maschine mit 25 Zuständen kompiliert. Es werden 10 Befehle übersprungen, wofür eine Sequenz von 9 Zuständen nötig ist.

```
goto MID;
halt; halt; halt; halt; halt;
MID:
goto END;
halt; halt; halt; halt; halt;
END:
```

Die Maschine zu diesem Programm hat nur noch 21 Zustände. Es werden zwei Mal 5 Zustände übersprungen, wofür jeweils dieselbe Sequenz mit 4 Zuständen genutzt wird. Die Befehlsauswahl-Submaschine bekommt einen Zustand dazu, weil das Programm nun einen Befehl mehr enthält. Trotzdem spart dies insgesamt vier Zustände.

### Vergleich zu naiven Aufbau

Der *naive* Aufbau bezeichnet den Ansatz die Maschine als Folge von befehlsausführenden Submaschinen aufzubauen, sodass jeder Befehl im Programm eine eigene Submaschine bekommt. Die Abschätzung der Größenordnung von naiv gebauten Maschinen kann in dieser Arbeit nicht exakt erfolgen, da dies nicht behandelt wurde. Der Nachteil der naiven Vorgehensweise ist, dass neben den statischen Submaschinen, auch die dynamischen Adressierung-Submaschinen für jeden Befehl, der auf Variablen arbeitet, einzeln eingebaut werden müssen. Daher kann die die Größe der Maschine durch $O(B \cdot V)$ abgeschätzt werden, was quadratisch und damit asymptotisch schlechter ist, als der Ansatz, der für diese Arbeit gewählt wurde.

## 6.6 Testen von kompilierten Maschinen

Bei der Entwicklung des Compilers war es essentiell, dass die erzeugte Maschine *exakt* das macht, was vom Quellcode vorgegeben wird.

Im Gegensatz zum Simulator gibt ein erfolgreicher Durchlauf des Programms keine Auskunft darüber, ob die kompilierte Maschine dem Quellcode entspricht. Der Simulator konnte einfach getestet werden, indem Maschinen simuliert wurden und das Ergebnis

mit dem erwarteten Verhalten abgeglichen wurde. Beispielsweise wurden auf diese Weise die verschiedene Busy Beaver zum Testen genutzt[13].

Um den Compiler zu testen, wurden kompilierte Maschinen simuliert und ihr Verhalten mit dem Quellcode abgeglichen. Für die Simulation wurde ein grafischer Online-Simulator verwendet, der unter https://turingmachine.io frei zugänglich ist.

Dieser spezifische Online-Simulator wurde gewählt, weil sowohl das Band als auch die Maschine grafisch dargestellt werden und eine textbasierte Beschreibungssprache für die Maschine genutzt wird. Für die Übersetzung von kompilierten Maschinen in die Beschreibungssprache wurde ein Python-Skript geschrieben.

Das Folgende ist ein Beispielprogramm in BusyC. Abbildung 6.11 zeigt einen Screenshot der dazugehörige Maschine im grafischen Online-Simulator.

```
uint a = 1;

LOOP:
a++;
goto LOOP;
```

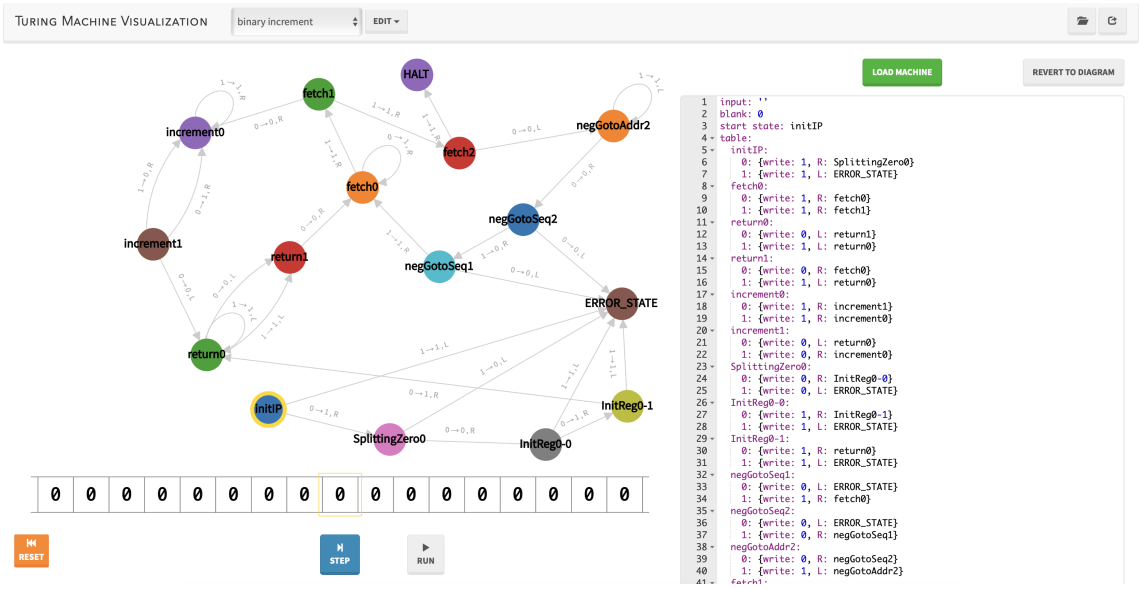



Abbildung 6.11: Der Online-Turingmaschinen-Simulator bei der Ausführung der Beispielmaschine. auf der linken Seite ist die Maschine abgebildet, auf der rechten Seite der Code für die Beschreibung der Maschine. Der Simulator hat einen Fehler, bei dem für gewisse Kanten der Pfeil auf der falschen Seite angezeigt wird. Die Simulation funktioniert einwandfrei. Screenshot erstellt am 17. Juli 2025.

[13] Im Beispiel des Busy Beavers von 5 wurden 47 176 870 Schritte und 4 098 Einsen auf dem Band erwartet.

Mit diesem Simulator wurden alle Submaschinen und viele Programme Schritt für Schritt getestet und verifiziert, dass die Maschine sich wie erwartet verhält. Dabei wurden alle denkbaren Randfälle ausprobiert.

# 7

# Mathematische Sätze auf Busy-Beaver-Werte reduzieren

Dieses Kapitel baut auf der Arbeit der letzten Kapitel auf und nutzt BusyC und den Compiler, um zwei offene mathematische Probleme auf Werte der Busy-Beaver-Funktion zu reduzieren. Die in diesem Kapitel behandelten Probleme sind die Frage, ob es eine ungerade perfekte Zahl gibt und die Goldbach-Vermutung.

### Reduktion von mathematischen Sätzen auf Busy-Beaver-Werte

Das Ziel dieses Kapitels ist **nicht** das lösen der beiden Probleme. Das Ziel ist die Reduktion auf Werte der BusyBeaverFunktion. Die Reduktion eines Satzes auf einen Busy-Beaver–Wert kann (und wird hier) mittels einer Turingmaschine passieren, die genau dann anhält wenn der Satz wahr ist oder falsch.

Sei $n$ die Anzahl der Zustände einer Turingmaschine $M$, die genau dann anhält, wenn Satz $A$ wahr oder falsch ist. Nach Definition der Busy-Beaver-Funktion gilt, dass $M$ nach maximal $BB(n)$ Schritten anhält, oder niemals anhält. Läuft $M$ länger als $BB(n)$ Schritte oder hält sie vorher an, lässt sich daraus ableiten, ob der Satz wahr oder falsch ist.

## 7.1 Ungerade Perfekte Zahl

Definition 7.1 (Perfekte Zahl). Eine Zahl $n \in \mathbb{N}$ wird *perfekte Zahl* genannt, wenn sie gleich der Summe ihrer positiven Teiler außer sich selbst ist.

Die ersten drei perfekten Zahlen sind $6$[14], $28$[15] und $496$[16]. Bisher sind nur gerade perfekte Zahlen bekannt und die Existenz ungerader perfekten Zahl ist ein offenes Problem in der Mathematik.

Satz 7.2. *Es existiert keine ungerade perfekte Zahl.*

[14] $6 = 1 + 2 + 3$
[15] $28 = 1 + 2 + 4 + 7 + 14$
[16] $496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$

Für die Reduktion des Satzes wurde ein Algorithmus in BusyC implementiert, der alle ungeraden natürlichen Zahlen ab 3[17] auf „Perfektheit“ prüft und anhält, sobald eine diese Eigenschaft erfüllt. Zur Veranschaulichung wird der Algorithmus zunächst in C vorgestellt.

```c
int main() {
    int n = 1;
    int sum;
    while (1) {
        n += 2;
        sum = 0;
        for (int i = 1; i < n; i++) {
            if (n % i == 0) sum += i;
        }
        if (n == sum) break; // perfekte Zahl gefunden
    }
    return 0;
}
```

Der Algorithmus ist nicht auf Laufzeit optimiert, sondern auf Simplizität, damit die später erzeugte Maschine möglichst klein ist. Es wird zum Beispiel in der *for*-Schleife bis $n-1$ iteriert statt bis $\left\lfloor \frac{n}{3} \right\rfloor$, obwohl ungerade Zahlen zwischen diesen Werten keine Teiler haben, weil dafür weniger Programmlogik benötigt wird. Es folgt nun die Implementierung des Algorithmus in BusyC. Dabei wurde dieses C-Programm so gut wie möglich in BusyC übersetzt. Der Code wird in den Hauptteil und die Unterprogramme aufgeteilt.

## Der Hauptteil

```
uint n = 1;
uint sum = 0;

MAIN:
  n++; n++;
  SET_SUM_ZERO: / Setze sum=0
  sum--;
  if (sum != 0) goto SET_SUM_ZERO;

  uint i = 0;
  SET_I_ZERO:
  i--;
  if (i != 0) goto SET_I_ZERO;
  i++; / Setze i=1 für den start der Schleife
  /for (int i = 1; i < n; i++)
  FOR_1_TO_N_MINUS_1:
    uint iIsDivisor = 0; / Wird auf 1 gesetzt, falls i ein Teiler von n ist
    goto IF_I_IS_DIVISOR; / n % i == 0 ?
    RETURN_IF_I_IS_DIVISOR:
    if (iIsDivisor != 0) goto ADD_I_TO_SUM; / if (n % i == 0) sum += i
```

[17] 1 ist keine perfekte Zahl, da ihr einziger Teiler sie selbst ist.

```
21      RETURN_ADD_I_TO_SUM:
22      iIsDivisor--;
23
24      i++; / Inkrementiere i für nächste Iteration
25      uint iLessN = 0;
26      iLessN--;
27      goto IF_I_LESS_N; / i < n ?
28      RETURN_IF_I_LESS_N:
29      if (iLessN != 0) goto FOR_1_TO_N_MINUS_1; / Springe zu Schleifenbeginn
            falls i < n
30
31    goto IF_SUM_EQ_N; / n == sum ?
```

Dieser Codeabschnitt zeigt die Hauptschleife des Programms. Am Anfang wird *n* um zwei erhöht, *sum* wird auf 0 gesetzt und *i* wird für die *for*-Schleife auf 1 gesetzt. Das Programm springt in das Unterprogramm *IF_I_IS_DIVISOR*, um zu prüfen, ob *i* ein Teiler von *n* ist. Ist dies der Fall, wird im Unterprogramm die Variable *iIsDivisor* auf 1 gesetzt.

Als nächstes wird in das Unterprogramm *ADD_I_TO_SUM* gesprungen, falls *iIsDivisor* im letzten Schritt auf 1 gesetzt wurde. Dieses Unterprogramm addiert den Wert von *i* zu *sum*. Danach wird *iIsDivisor* für die nächste Iteration dekrementiert, auch wenn es schon den Wert 0 hatte. In dem Fall bleibt der Wert erhalten[18].

Am Ende der *for*-Schleife wird die nächste Iteration vorbereitet. Dafür wird *i* inkrementiert und in das Unterprogramm *IF_I_LESS_N* gesprungen, welches prüft, ob *i* kleiner als *n* ist und das Ergebnis in der Variable *iLessN* speichert. Falls *i* kleiner als *n* ist, springt das Programm zum Anfang der *for*-Schleife zurück. Andernfalls wird diese verlassen.

Nach der *for*-Schleife springt das Programm zum Unterprogramm *IF_SUM_EQ_N*, welches prüft, ob *n* gleich *sum* (die Summe der Teiler von *n* außer *n* selbst) ist. Wenn die Eigenschaft erfüllt ist, hält das Programm an. Wenn nicht, springt es zum Label *MAIN* zurück und Programm prüft die nächste ungerade Zahl auf „Perfektheit“.

### IF_I_IS_DIVISOR

Die vier Unterprogramme nutzen zwei Variablen als Puffer für die Variablen *n* und *i*. Diese werden hier einmal deklariert.

```
1  uint nBuf = 0;
2  uint iBuf = 0;
```

Dieser Codeausschnitt zeigt das Unterprogramm *IF_I_IS_DIVISOR*, welches im Hauptteil aufgerufen wird, um zu prüfen, ob die Variable *i* ein Teiler der Variable *n* ist.

```
1  IF_I_IS_DIVISOR: /Ziehe i so oft von n ab, bis n=0
2      if (n == 0) goto EVAL_IF_I_DIVISOR; / Falls n=0, prüfe ob auch i=0
3      if (i == 0) goto RESTORE_I_2; / Falls i=0, stelle den Wert wieder her
4      n--; i--;
5      nBuf++; iBuf++;
6      goto IF_I_IS_DIVISOR;
7
```

[18] Siehe Kapitel 4

```
  EVAL_IF_I_DIVISOR:
    if (i == 0) iIsDivisor++; / Falls n=0 und i=0 ist i ein Teiler von n
    RESTORE_N:
    n++; nBuf--;
    if (nBuf != 0) goto RESTORE_N;
    RESTORE_I:
    i++; iBuf--;
    if (iBuf != 0) goto RESTORE_I;
    goto RETURN_IF_I_IS_DIVISOR;

  RESTORE_I_2:
    i++; iBuf--;
    if (iBuf != 0) goto RESTORE_I_2;
    goto IF_I_IS_DIVISOR;
```

Die Variablen *n* und *i* werden so lange dekrementiert, bis *n* den Wert 0 hat. Sobald *i* den Wert 0 hat, springt das Programm in ein Unterprogramm, welches den ursprünglichen Wert von *i* wieder herstellt mithilfe von *iBuf*. Danach springt das Programm wieder zurück in die Schleife, in der *n* und *i* dekrementiert werden. Sobald *n* den Wert 0 hat, springt das Programm zum Unterprogramm *EVAL_IF_I_DIVISOR*, in dem die Variable *iIsDivisor* auf 1 gesetzt wird, falls *i* den Wert 0 hat. Der Wert von *i* zu diesem Zeitpunkt ist der Rest der ganzzahligen Division von *n* geteilt durch *i*. Ist dieser Rest 0, ist *i* ein Teiler von *n*.

Danach werden die ursprünglichen Werte von *n* und *i* wiederhergestellt und das Programm springt in den Hauptteil zurück.

### ADD_I_TO_SUM

```
ADD_I_TO_SUM: / Addiere i zur Summe
    sum++; iBuf++; i--;
    if (i != 0) goto ADD_I_TO_SUM;

  RESTORE_I_3:
    i++; iBuf--;
    if (iBuf != 0) goto RESTORE_I_3;
  goto RETURN_ADD_I_TO_SUM;
```

Dieses Unterprogramm addiert den Wert von *i* auf *sum*, indem *i* in einer Schleife dekrementiert wird und *sum* inkrementiert wird, bis *i* den Wert 0 hat. Danach wird der ursprüngliche Wert von *i* mithilfe von *iBuf* wieder hergestellt und das Programm kehrt in den Hauptteil zurück.

### IF_I_LESS_N

```
/ Es gilt hier immer n>=i, da der initiale Wert von n größer als der
    initiale Wert von i is
/ i wird in jeder Iteration inkrementiert bis i=n gilt, dann endet die
    "for-Schleife"
IF_I_LESS_N: / Zähle n und i runter bis i=0
```

```
  n--; i--; iBuf++;
  if (i != 0) goto IF_I_LESS_N;
  if (n != 0) iLessN++; / Setze auf 1, falls n>i
  RESTORE_N_AND_I: / Stelle die ursprünglichen Werte von n und i wieder her
    n++; i++; iBuf--;
    if (iBuf != 0) goto RESTORE_N_AND_I;
  goto RETURN_IF_I_LESS_N;
```

Dieses Unterprogramm prüft, ob *i* echt kleiner als *n* ist. Dafür dekrementiert es *n* und *i*, bis *i* den Wert 0 hat. Falls *n* am Ende auch den Wert 0 hat, war *i* nicht kleiner als *n*. Für die Wiederherstellung von *n* und *i* werden beide um den ursprünglichen Wert von *i* erhöht. Damit dies korrekt funktioniert ist wichtig, dass *n* vor Beginn des Unterprogramms größer gleich *i* ist. Dies ist immer der Fall, da der initiale Wert von *i* in der *for*-Schleife 1 ist, und *n* bei 3 startet. Nach jedem Inkrementieren von *i* wird dieses Unterprogramm aufgerufen und sobald *i=n* gilt, wird die *for*-Schleife verlassen.

### IF_SUM_EQ_N

```
uint bothZero = 0;
IF_SUM_EQ_N: / Zähle n und sum runter, bis eine der Variablen den Wert 0 hat
  if (sum == 0) goto EVAL_SUM_EQ_N;
  sum--; n--; nBuf++;
  if (n != 0) goto IF_SUM_EQ_N;

  EVAL_SUM_EQ_N:
    bothZero++; bothZero++;
    if (sum == 0) bothZero--;
    if (n == 0) bothZero--;
    if (bothZero == 0) halt; / Wenn n und sum am Ende 0 sind, wurde eine
        perfekte Zahl gefunden
    RESTORE_N_2:
      n++; nBuf--;
      if (nBuf != 0) goto RESTORE_N_2;
    bothZero--; / Setze auf 0 für das nächste n
    goto MAIN;
```

Dieses letzte Unterprogramm prüft, ob *n* eine perfekte Zahl ist. Dafür werden *n* und *sum* dekrementiert, bis eine der beiden Variablen den Wert 0 hat. Danach prüft das Programm, ob beide den Wert 0, was bedeutet, dass beide den gleichen ursprünglichen Wert hatten. Falls dies zutrifft, hält das Programm an. Andernfalls wird der Wert von *n* wiederhergestellt und das Programm springt zum Anfang des Hauptteils zurück, um die nächste ungerade Zahl zu prüfen.

### Die Größe der erzeugten Turingmaschine

Fügt man alle Codeschnipsel dieses Programms in der vorgestellten Reihenfolge in eine Datei ein und kompiliert diese, erhält man eine Turingmaschine mit 297 Zuständen. Durch

weitere Optimierungen im Code konnte die Anzahl der Zustände auf **222** Zustände gesenkt werden. Die folgenden Optimierungen wurden vorgenommen:

- Einfügen von *goto*-Befehlen, um die Sprungsequenzen zu kürzen[19]
- Ändern der Reihenfolge der Variablendefinitionen
- Zusammenfassen der Variablen *iIsDivisor*, *iLessN* und *bothZero* zu einer Variable *bool*

Das vollständige optimierte Programm befindet sich im Anhang der Arbeit.

Die erzeugte Maschine reduziert den Satz „Es gibt keine ungerade perfekte Zahl“ auf $BB(222)$. Bei einer Literaturrecherche wurde keine frühere Reduktion dieses Satzes oder seiner Negation auf einen Wert der Busy-Beaver-Funktion gefunden.

## 7.2 Die Goldbach-Vermutung

Satz 7.3 (Goldbach-Vermutung). *Jede gerade Zahl größer als 2, ist die Summe zweier Primzahlen.*

Die Goldbach-Vermutung ist ein weiteres ungelöstes Problem in der Mathematik. Für die Reduktion wurde ein Algorithmus in BusyC implementiert, der über alle geraden Zahlen ab 4 iteriert und prüft, ob sie die Summe zweier Primzahlen ist. Wie im vorherigen Abschnitt wird der Algorithmus zuerst in C präsentiert. Danach folgt die Implementierung in BusyC.

```
int isPrime(int p) {
  int numDivisors = 0;
  for (int j = 1; j < p; j++) {
    if (p % j == 0) numDivisors++;
  }
  return (numDivisors == 1);
}

int main() {
  int n = 2;
  int isSumOf2Primes = 0;
  while (1){
    n += 2;
    isSumOf2Primes = 0;
    for (int i = 2; i < n-1; i++) {
      if (isPrime(i) && isPrime(n-i)) isSumOf2Primes = 1;
    }
    if (isSumOf2Primes == 0) return 0;
  }
  return 0;
}
```

Wie im vorherigen Abschnitt ist das BusyC-Programm in Hauptteil und Unterprogramme aufgeteilt.

[19] Diese Optimierung kann sicherlich weiter ausgereizt werden, um die Maschine noch kleiner zu machen.

### Hauptteil

```
uint n = 2;
uint isSumOf2Primes = 0;

MAIN:
  n++;n++;
  SET_IS_SUM_OF_2_PRIMES_ZERO:
    isSumOf2Primes--;
    if (isSumOf2Primes != 0) goto SET_IS_SUM_OF_2_PRIMES_ZERO;

  uint i = 0;
  SET_I_ZERO:
    i--;
    if (i != 0) goto SET_I_ZERO;
  i++; i++; / Setze i=2 für die Schleife
  FOR_2_TO_N_MINUS_2:
    uint prime = 0;
    goto IF_SUM_OF_2_PRIMES; / if (isPrime(i) && isPrime(n-i))
    RETURN_IF_SUM_OF_2_PRIMES:
    if (prime != 0) isSumOf2Primes++; / Falls i und n-i prim
    prime--;

    i++;
    uint iLessNm1 = 0;
    iLessNm1--;
    n--;
    goto IF_I_LESS_N_MINUS_1; /Prüfe ob i<n-1
    RETURN_IF_I_LESS_N_MINUS_1:
    n++;
    if (iLessNm1 != 0) goto FOR_2_TO_N_MINUS_2;
  / Halte falls n nicht summe zweier Primzahlen ist
  if (isSumOf2Primes == 0) halt;
  goto MAIN;
```

Das Programm prüft für alle geraden Zahlen $n \geq 4$ in einer *for*-Schleife mit der Laufvariable $i$, die von 2 bis $n-2$ läuft, ob $i$ und $n-i$ Primzahlen sind. Sollte dies in der *for*-Schleife nicht vorkommen, hält das Programm an.

### IF_SUM_OF_2_PRIMES

```
uint nBuf = 0;
uint iBuf = 0;
uint p = 0;
IF_SUM_OF_2_PRIMES:
  i--; p++; iBuf++; / Schreibe Wert von i in p
  if (i != 0) goto IF_SUM_OF_2_PRIMES;
  goto IS_PRIME; / Prüfe ob i prim ist

```

```
  RETURN_I_PRIME:
  n--; p++; nBuf++; / Schreibe Wert von n in p
  if (n != 0) goto RETURN_I_PRIME;

  SUBTRACT_I:
  iBuf--; p--; i++; / Subtrahiere i von p und stelle i wieder her
  if (iBuf != 0) goto SUBTRACT_I;
  goto IS_PRIME; / Prüfe on n-i prim ist
  RETURN_N_MINUS_I_PRIME:
  n++; nBuf--; / Stelle n wieder her
  if (nBuf != 0) goto RETURN_N_MINUS_I_PRIME;
  prime--;
  goto RETURN_IF_SUM_OF_2_PRIMES;
```

Dieses Unterprogramm prüft, ob $i$ und $n - i$ Primzahlen sind. Dafür ruft es zwei Mal das Unterprogramm *IS_PRIME* auf, welches prüft, ob $p$ eine Primzahl ist. Vor den Aufrufen wird $p$ auf den Wert von $i$ bzw. $n - i$ gesetzt. *IS_PRIME* inkrementiert die Variable *prime*, falls $p$ prim war.

Am Ende dieses Unterprogramms wird *prime* dekrementiert. Dadurch hat es den Wert 0, falls höchstens einer der Werte $i$ oder $n - i$ prim war. Waren beide prim, hat *prime* nun den Wert 1.

### IS_PRIME

```
uint pBuf = 0;
uint jBuf = 0;
uint numDivisors = 0;
uint jIsDivisor = 0;
IS_PRIME:
  numDivisors--;
  if (numDivisors != 0) goto IS_PRIME; / Setze numDivisors=0

  uint j = 0;
  SET_J_ZERO:
  j--;
  if (j != 0) goto SET_J_ZERO;
  j++;
  FOR_1_TO_P_MINUS_1:
    goto IF_J_IS_DIVISOR;
    RETURN_IF_J_IS_DIVISOR:
    if (jIsDivisor != 0) numDivisors++;
    jIsDivisor--;

    j++;
    uint jLessP = 0;
    jLessP--;
    goto IF_J_LESS_P;
    RETURN_IF_J_LESS_P:
    if (jLessP != 0) goto FOR_1_TO_P_MINUS_1;
```

```
26
27    numDivisors--;
28    SET_P_ZERO:
29      p--;
30      if (p != 0) goto SET_P_ZERO;
31    if (numDivisors == 0) prime++;
32    / Gehe dahin zurück wo aufgerufen wurde
33    if (nBuf == 0) goto RETURN_I_PRIME;
34    goto RETURN_N_MINUS_I_PRIME;
35
36    IF_J_LESS_P:
37      p--; j--; jBuf++;
38      if (j != 0) goto IF_J_LESS_P;
39      if (p != 0) jLessP++;
40      RESTORE_P_AND_J:
41        p++; j++; jBuf--;
42        if (jBuf != 0) goto RESTORE_P_AND_J;
43      goto RETURN_IF_J_LESS_P;
44
45
46    IF_J_IS_DIVISOR:
47      if (p == 0) goto EVAL_IF_J_DIVISOR;
48      if (j == 0) goto RESTORE_J_2;
49      p--; j--; pBuf++; jBuf++;
50      goto IF_J_IS_DIVISOR;
51
52      EVAL_IF_J_DIVISOR:
53        if (j == 0) jIsDivisor++;
54        RESTORE_P:
55          p++; pBuf--;
56          if (pBuf != 0) goto RESTORE_P;
57        RESTORE_J:
58          j++; jBuf--;
59          if (jBuf != 0) goto RESTORE_J;
60        goto RETURN_IF_J_IS_DIVISOR;
61
62      RESTORE_J_2:
63        j++; jBuf--;
64        if (jBuf != 0) goto RESTORE_J_2;
65        goto IF_J_IS_DIVISOR;
```

Dieses Unterprogramm prüft, ob $p$ eine Primzahl ist. Dafür nutzt es die Laufvariable $j$ um von 1 bis $p-1$ zu iterieren, wobei es prüft, ob $j$ ein Teiler von $p$ ist. In der Variable *numDivisors* wird die Anzahl der Teiler gespeichert. Der Prüfen, ob $j$ ein Teiler von $p$ ist, funktioniert so wie beim Programm im letzten Abschnitt. Da von 1 bis $p-1$ iteriert wird, wird *numDivisors* bei Primzahlen nur ein Mal inkrementiert (1 ist der einzige Teiler kleiner als $p$). Bei nicht-Primzahlen passiert dies öfter. Am Ende wird *numDivisors* dekrementiert und *prime* inkrementiert, falls *numDivisors* den Wert 0 hat.

Da dieses Unterprogramm zwei Mal aufgerufen wird, muss ermittelt werden, wohin

am Ende zurückgesprungen werden muss. Dies passiert mithilfe der Variable *nBuf*. Diese hat den Wert 0 während geprüft wird, ob $i$ prim ist. Beim nächsten Aufruf, bei dem geprüft wird ob $n - i$ prim ist, hat *nBuf* den Wert von $n$.

### IF_I_LESS_N_MINUS_1

```
1 IF_I_LESS_N_MINUS_1: / Zähle n und i runter bis i=0
2   n--; i--; iBuf++;
3   if (i != 0) goto IF_I_LESS_N_MINUS_1;
4   if (n != 0) iLessNm1++; / Setze auf 1, falls n-1>i
5   RESTORE_N_AND_I: / Stelle die ursprünglichen Werte von n und i wieder her
6     n++; i++; iBuf--;
7     if (iBuf != 0) goto RESTORE_N_AND_I;
8   goto RETURN_IF_I_LESS_N_MINUS_1;
```

Dieses Unterprogramm, prüft ob $i$ kleiner als $n - 1$ ist. Es funktioniert wie das Unterprogramm *IF_I_LESS_N* aus dem letzten Abschnitt mit dem Unterschied, dass vor dem Aufruf $n$ dekrementiert wird, damit $i < n - 1$ und nicht $i < n$ geprüft wird. Nach der Rückkehr in den Hauptteil wird $n$ inkrementiert.

### Die Größe der erzeugten Turingmaschine

Das, durch Zusammenfügen der Codeschnipsel in der gegebenen Reihenfolge, erzeugte Programm wird in ein Turingmaschine mit 432 Zuständen kompiliert. Die Anzahl der Zustände kann durch Optimierungen wieder gesenkt werden, sodass das optimierte Programm eine Maschine mit **290** Zuständen erzeugt. Die folgenden Optimierungen wurden vorgenommen.

- Einfügen von *goto*-Befehlen, um die Sprungsequenzen zu kürzen
- Ändern der Reihenfolge der Variablendefinitionen
- Zusammenfassen der Variablen iLessNm1 und jLessP zu einer Variable bool
- Verschieben des Unterprogramms *IF_I_LESS_N* direkt unter den Hauptteil

Das vollständige optimierte Programm befindet sich im Anhang der Arbeit.

Die erzeugte Maschine reduziert die Goldbach-Vermutung auf $BB(290)$ und benötigt damit 94% weniger Zustände als die 4 888-Zustands-Maschine aus [9]. Durch manuelle Konstruktion kann eine äquivalente Maschine mit 27 Zuständen erzeugt werden [4].

# 8

# Zusammenfassung

Im Rahmen dieser Arbeit wurde ein schneller Turingmaschinen-Simulator entwickelt, der die Maschine blockweise simuliert. Im Optimalfall simuliert er eine Maschine mit weniger als einem CPU-Takt pro Maschinenschritt. Außerdem wurden die Programmiersprache BusyC und ein Compiler entwickelt, der ein BusyC-Programm in eine Turingmaschine übersetzt. Die Anzahl der Zustände der erzeugten Maschine ist dabei linear relativ zur Größe des Programms. Mithilfe von BusyC und dem Compiler wurden zwei Maschinen konstruiert, die jeweils genau dann halten, wenn es eine ungerade perfekte Zahl gibt (diese Maschine hat 222 Zustände), und wenn die Goldbach-Vermutung falsch ist (290 Zustände). Letztere besitzt 94% weniger Zustände, als die äquivalente Maschine aus der Arbeit *A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory* [9].

## 8.1 Verworfene Ansätze

Während der Entwicklung des Simulators wurde ausprobiert, den gesamten Speicher vor zu berechnen, anstatt dies dynamisch während der Simulation zu machen. Die Idee dahinter ist, dass eine *if* -Bedingung gespart werden kann, da man nicht mehr prüfen muss, ob der Speicher für eine Konfiguration schon einen Wert hat. Diese Funktion wurde vollständig implementiert. Dazu gehörte auch das Prüfen, ob die Maschine auf einem Block in eine Endlosschleife gerät.

In der Praxis konnte beim Busy-Beaver von 5 kein Geschwindigkeitsunterschied festgellt werden, vermutlich wegen Branch Prediction der CPU. Das Vorberechnen des gesamten Speichers ist außerdem eine massive Aufwandsverschwendung, weil bei allen getesteten Maschinen[20] die meisten Konfiguration nicht erreicht werden. Außerdem wächst die Größe des Speichers exponentiell bezüglich zur Blockgröße, daher ist dieser Ansatz nur für kleinere Blockgrößen anwendbar.

Der Compiler wurde ursprünglich so implementiert, dass der Programmzähler runterzählt. Dazu musste dieser zur Anzahl der Befehle initalisiert werden, was die erzeugte Maschine signifikant größer gemacht hat. Der finale Compiler lässt den Programmzähler hochzählen, sodass dieser bei 0 startet.

[20] Busy-Beaver für 2, 4 und 5, sowie Zahlreiche Maschinen, die mit dem Compiler erzeugt wurden.

Diese Änderung zu implementieren war etwas „fummelig“, da die Befehlsauswahl–Submaschine, sowie die Submaschinen für Sprünge geändert werden mussten.

## 8.2 Weitere Verbesserungen

Da der Großteil Speicher des Simulators in der Praxis sehr leer bleibt, könnte eine Hash–Funktion genutzt werden, um den Speicherbedarf zu reduzieren. Dadurch könnten größere Blockgrößen verwendet werden. Diese kann nicht beliebig groß werden, sondern hängt von der Busbreite der CPU ab.

Für die Hash-Funktion wäre außerdem wichtig, dass diese sehr schnell berechnet werden kann. Sonst kann man sich das auch sparen und eine kleinere Blockgröße verwenden.

BusyC könnte druch *Syntactic Sugar* erweitert werden, um das Programmieren zu erleichtern und den Code lesbarer zu machen. Praktische neue Funktionen wären Schleifen und direkte Wertezuweisungen für Variablen. Schleifen könnten mithilfe von bedingten Sprüngen realisiert werden (siehe Codeausschnitt 8.2). Direkte Wertezuweisungen für Variablen könnten durch „SET_ZERO“-Schleifen und Inkrement-Befehlen realisiert werden (siehe Codeausschnitt 8.2).

```
while (b != 0) {...} / Syntactic Sugar

/ Interne Realisierung
WHILE:
if (b == 0) goto AFTER_WHILE;
...
goto WHILE;
AFTER_WHILE:
```

Abbildung 8.1: Die Realisierung einer *while*-Schleife mithilfe von *Syntactic Sugar*.

```
b = 4; / Syntactic Sugar

/ Interne Realisierung
SET_B_ZERO:
b--;
if (b != 0) goto SET_B_ZERO;
b++; b++; b++; b++;
```

Abbildung 8.2: Die Realisierung einer Wertezuweisungen mithilfe von *Syntactic Sugar*.

# Literatur


[1] Aaronson, S. The Busy Beaver Frontier. In: *SIGACT News* 51(3):32–54, Sep. 2020. Archived from the original on 5 July 2022. ISSN: 0163-5700. DOI: 10.1145/3427361.3427369. URL: https://www.scottaaronson.com/papers/bb.pdf.

[2] Barak, B. *Introduction to Theoretical Computer Science*. Online manuscript. 2014. URL: https://introtcs.org.

[3] BB Challenge Community *We Have Proved BB(5) = 47,176,870*. https://discuss.bbchallenge.org/t/july-2nd-2024-we-have-proved-bb-5-47-176-870/237. Forum post describing the proof and linking to the formal Coq proof. Accessed: 2025-08-12. 2024.

[4] Leng, Y. *GitHub Repository "goldbach_tm27"*. https://github.com/lengyijun/goldbach_tm. Accessed: 2025-08-13. 2024.

[5] mxdys *BB(6) Lower Bound*. https://wiki.bbchallenge.org/wiki/1RB1RA_1RC1RZ_1LD0RF_1RA0LE_0LD1RC_1RA0RE. Accessed: 2025-08-12. Juni 2025.

[6] Rado, T. On Non-Computable Functions. In: *The Bell System Technical Journal* 41(3):877–884, 1962. DOI: 10.1002/j.1538-7305.1962.tb00480.x.

[7] Riebel, J. The Undecidability of BB(748): Understanding Gödel's Incompleteness Theorems. Archived from the original (PDF) on 17 September 2024. Retrieved 24 September 2024. Bachelor's thesis. University of Augsburg, März 2023. URL: https://www.ingo-blechschmidt.eu/assets/bachelor-thesis-undecidability-bb748.pdf.

[8] Sipser, M. *Introduction to the Theory of Computation*. 3rd. Cengage Learning, 2013.

[9] Yedidia, A. und Aaronson, S. *A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory*. 2016. arXiv: 1605.04343 [cs.LO]. URL: https://arxiv.org/abs/1605.04343.

# A

# Optimierte BusyC-Programme

## A.1 Ungerade Perfekte Zahlen

```
uint n = 1;
uint i = 0;
uint sum = 0;
uint bool = 0; / Ersetzt 3 Variablen
uint nBuf = 0;
uint iBuf = 0;


MAIN:
  n++; n++;
  SET_SUM_ZERO:
  sum--;
  if (sum != 0) goto SET_SUM_ZERO;

  SET_I_ZERO:
  i--;
  if (i != 0) goto SET_I_ZERO;
  i++;
  FOR_1_TO_N_MINUS_1:
    bool--; / Dekrementiere hier, da am Ende einer Iteration inkrementiert
        wird
    goto IF_I_IS_DIVISOR;
    RETURN_IF_I_IS_DIVISOR:
    if (bool != 0) goto ADD_I_TO_SUM_0;
    RETURN_ADD_I_TO_SUM:
    bool--;

    i++;
    goto IF_I_LESS_N_0;
    RETURN_IF_I_LESS_N:
    if (bool != 0) goto FOR_1_TO_N_MINUS_1;

  goto IF_SUM_EQ_N_0;
```

```
33
34
35
36 IF_I_IS_DIVISOR:
37     if (n == 0) goto EVAL_IF_I_DIVISOR;
38     if (i == 0) goto RESTORE_I_2;
39     n--; i--;
40     nBuf++; iBuf++;
41     goto IF_I_IS_DIVISOR;
42
43 MAIN_2:
44 goto MAIN;
45
46   EVAL_IF_I_DIVISOR:
47     if (i == 0) bool++;
48     RESTORE_N:
49     n++; nBuf--;
50     if (nBuf != 0) goto RESTORE_N;
51     RESTORE_I:
52     i++; iBuf--;
53     if (iBuf != 0) goto RESTORE_I;
54     goto RETURN_IF_I_IS_DIVISOR;
55
56 / Zum Kürzen der Sprungsequenz
57 ADD_I_TO_SUM_0:
58 goto ADD_I_TO_SUM;
59
60 RETURN_ADD_I_TO_SUM_0:
61 goto RETURN_ADD_I_TO_SUM;
62
63   RESTORE_I_2:
64     i++; iBuf--;
65     if (iBuf != 0) goto RESTORE_I_2;
66     goto IF_I_IS_DIVISOR;
67
68
69
70 / Zum Kürzen der Sprungsequenzen
71 IF_I_LESS_N_0:
72 goto IF_I_LESS_N;
73
74 MAIN_1:
75 goto MAIN_2;
76
77 RETURN_IF_I_LESS_N_0:
78 goto RETURN_IF_I_LESS_N;
79
80 IF_SUM_EQ_N_0:
81 goto IF_SUM_EQ_N;
82
```

```
83
84  ADD_I_TO_SUM:
85      sum++; iBuf++; i--;
86      if (i != 0) goto ADD_I_TO_SUM;
87
88    RESTORE_I_3:
89      i++; iBuf--;
90      if (iBuf != 0) goto RESTORE_I_3;
91    goto RETURN_ADD_I_TO_SUM_0;
92
93  MAIN_0:
94  goto MAIN_1;
95
96  IF_I_LESS_N:
97    n--; i--; iBuf++;
98    if (i != 0) goto IF_I_LESS_N;
99    if (n != 0) bool++;
100   RESTORE_N_AND_I:
101     n++; i++; iBuf--;
102     if (iBuf != 0) goto RESTORE_N_AND_I;
103   goto RETURN_IF_I_LESS_N_0;
104
105
106
107 IF_SUM_EQ_N:
108   if (sum == 0) goto EVAL_SUM_EQ_N;
109   sum--; n--; nBuf++;
110   if (n != 0) goto IF_SUM_EQ_N;
111
112   EVAL_SUM_EQ_N:
113     bool++; bool++;
114     if (sum == 0) bool--;
115     if (n == 0) bool--;
116     if (bool == 0) halt;
117     RESTORE_N_2:
118       n++; nBuf--;
119       if (nBuf != 0) goto RESTORE_N_2;
120     goto MAIN_0;
```

## A.2 Goldbach-Vermutung

```
1  uint p = 0;
2  uint j = 0;
3  uint isSumOf2Primes = 0;
4  uint nBuf = 0;
5  uint numDivisors = 0;
6  uint n = 2;
7  uint i = 0;
```

```
uint bool = 0; / Ersetzt 2 Variablen
uint prime = 0;
uint iBuf = 0;
uint pBuf = 0;
uint jBuf = 0;
uint jIsDivisor = 0;

MAIN:
  n++;n++;
  SET_IS_SUM_OF_2_PRIMES_ZERO:
    isSumOf2Primes--;
    if (isSumOf2Primes != 0) goto SET_IS_SUM_OF_2_PRIMES_ZERO;

  SET_I_ZERO:
    i--;
    if (i != 0) goto SET_I_ZERO;
  i++; i++;
  FOR_2_TO_N_MINUS_2:
    goto IF_SUM_OF_2_PRIMES;
    RETURN_IF_SUM_OF_2_PRIMES:
    if (prime != 0) isSumOf2Primes++;
    prime--;

    i++;
    bool--;
    n--;
    goto IF_I_LESS_N_MINUS_1;
    RETURN_IF_I_LESS_N_MINUS_1:
    n++;
    if (bool != 0) goto FOR_2_TO_N_MINUS_2;

  if (isSumOf2Primes == 0) halt;
  goto MAIN;



IF_I_LESS_N_MINUS_1:
  n--; i--; iBuf++;
  if (i != 0) goto IF_I_LESS_N_MINUS_1;
  if (n != 0) bool++;
  RESTORE_N_AND_I:
    n++; i++; iBuf--;
    if (iBuf != 0) goto RESTORE_N_AND_I;
  goto RETURN_IF_I_LESS_N_MINUS_1;

/ Kürze Sprungsequenz
RETURN_IF_SUM_OF_2_PRIMES_0:
goto RETURN_IF_SUM_OF_2_PRIMES;


```

```
58 IF_SUM_OF_2_PRIMES:
59   i--; p++; iBuf++;
60   if (i != 0) goto IF_SUM_OF_2_PRIMES;
61   goto IS_PRIME;
62 
63   RETURN_I_PRIME:
64   n--; p++; nBuf++;
65   if (n != 0) goto RETURN_I_PRIME;
66 
67   SUBTRACT_I:
68   iBuf--; p--; i++;
69   if (iBuf != 0) goto SUBTRACT_I;
70   goto IS_PRIME;
71   RETURN_N_MINUS_I_PRIME:
72   n++; nBuf--;
73   if (nBuf != 0) goto RETURN_N_MINUS_I_PRIME;
74   prime--;
75   goto RETURN_IF_SUM_OF_2_PRIMES_0;
76 
77 / Kürze Sprungsequenz
78 RETURN_I_PRIME_0:
79 goto RETURN_I_PRIME;
80 
81 IS_PRIME:
82   numDivisors--;
83   if (numDivisors != 0) goto IS_PRIME;
84 
85   SET_J_ZERO:
86   j--;
87   if (j != 0) goto SET_J_ZERO;
88   j++;
89   FOR_1_TO_P_MINUS_1:
90     goto IF_J_IS_DIVISOR;
91     RETURN_IF_J_IS_DIVISOR:
92     if (jIsDivisor != 0) numDivisors++;
93     jIsDivisor--;
94 
95     j++;
96     bool--;
97     goto IF_J_LESS_P;
98     RETURN_IF_J_LESS_P:
99     if (bool != 0) goto FOR_1_TO_P_MINUS_1;
100 
101  numDivisors--;
102  SET_P_ZERO:
103    p--;
104    if (p != 0) goto SET_P_ZERO;
105  if (numDivisors == 0) prime++;
106 
107  if (nBuf == 0) goto RETURN_I_PRIME_0;
```

```
108   goto RETURN_N_MINUS_I_PRIME;
109
110 RETURN_IF_J_IS_DIVISOR_0:
111 goto RETURN_IF_J_IS_DIVISOR;
112
113   IF_J_LESS_P:
114     p--; j--; jBuf++;
115     if (j != 0) goto IF_J_LESS_P;
116     if (p != 0) bool++;
117     RESTORE_P_AND_J:
118       p++; j++; jBuf--;
119       if (jBuf != 0) goto RESTORE_P_AND_J;
120     goto RETURN_IF_J_LESS_P;
121
122
123   IF_J_IS_DIVISOR:
124     if (p == 0) goto EVAL_IF_J_DIVISOR;
125     if (j == 0) goto RESTORE_J_2;
126     p--; j--; pBuf++; jBuf++;
127     goto IF_J_IS_DIVISOR;
128
129     EVAL_IF_J_DIVISOR:
130       if (j == 0) jIsDivisor++;
131       RESTORE_P:
132         p++; pBuf--;
133         if (pBuf != 0) goto RESTORE_P;
134       RESTORE_J:
135         j++; jBuf--;
136         if (jBuf != 0) goto RESTORE_J;
137       goto RETURN_IF_J_IS_DIVISOR_0;
138
139     RESTORE_J_2:
140       j++; jBuf--;
141       if (jBuf != 0) goto RESTORE_J_2;
142       goto IF_J_IS_DIVISOR;
```

# B

# GitHub Repository

Der vollständige Quellcode des Simulators und des Compilers, sowie einige BusyC–Programme und weitere Ressourcen sind im folgenden GitHub Repository verfügbar:

https://github.com/MarvinDetzkeit/Bachelorarbeit