

# WHILE $\rightarrow$ TM

## Aufgabenstellung

In dieser Aufgabe sollen Sie die in der Vorlesung angedeutete Übersetzung von WHILE-Programmen zu deterministischen Mehrband-Turingmaschinen implementieren. Die genaue Spezifikation ist wie folgt:

Die folgende Tabelle zeigt, wie WHILE-Programme in Java realisiert wurden und wie die entsprechende Notation dazu aussieht:

Java-Syntax	Notation
Increment ( $x_i, x_j, c$ )	$x_i := x_j + c$
Decrement ( $x_i, x_j, c$ )	$x_i := x_j - c$
Block ( $p_1, \dots, p_k$ )	$p_1 ; \dots ; p_k$
IfThenElse ( $x_i, p_t, p_e$ )	IF $x_i = 0$ THEN $p_t$ ELSE $p_e$ END
While ( $x_i, p$ )	WHILE $x_i \neq 0$ DO $p$ END

Hierbei sind die  $x_i$  jeweils Variablen und die  $p...$  WHILE-Programme. Der einzige Unterschied zur Vorlesung ist, dass wir eine  $n$ -äre Komposition von Programmen namens Block definieren. Dieser besteht aus einer Liste von Programmen, die einfach nacheinander ausgeführt werden. Beachten Sie, dass diese Liste auch leer sein kann oder nur ein Programm enthalten kann.

Sei nun  $p$  ein WHILE-Programm und  $n$  der Index der größten im Programm vorkommenden Variable (oder  $-1$  falls gar keine Variable vorkommt). Dann definieren wir die Menge der im Programm verwendeten Variablen als  $\{x_0, \dots, x_n\}$  und repräsentieren den Zustand des Programms als einen Vektor von  $n + 1$  natürlichen Zahlen, nämlich der in den Variablen  $x_0, \dots, x_n$  gespeicherten Werte.

Wir übersetzen nun  $p$  in eine  $n + 1$ -Band-Turingmaschine über dem Alphabet  $\{X\}$ . Die Idee hierbei ist:

- Jede Variable hat ihr eigenes Band.
- Wenn die Variable  $x_i$  den Wert 0 hat, ist das Band  $i$  komplett leer.
- Wenn die Variable  $x_i$  einen Wert  $k > 0$  hat, steht auf dem Band  $i$  genau  $k$  mal das Zeichen  $X$  und der Schreib-/Lesekopf steht auf dem linken dieser Zeichen:

$$\dots \square \square \square \underbrace{X X \dots X}_{k \text{ mal}} \square \square \square \dots$$

↓

## Implementierungshinweise

Alle TM-Zustände, die Sie erzeugen, sind *immer* voneinander verschieden, auch wenn Sie Ihnen keine Namen oder mehreren den gleichen Namen geben. Sie müssen Ihren TM-Zuständen keine Namen geben, aber für Debugging-Zwecke kann es womöglich hilfreich sein.

**Achtung:** Diese Aufgabe ist vermutlich schwierig anhand von Testfällen zu debuggen. Es ist daher besonders wichtig, dass Sie Ihre Konstruktion und die Implementierung dafür so knapp und elegant wie möglich halten, um Fehlerquellen zu minimieren. Wir haben Ihnen dafür im Java-Template bereits einiges an Struktur zur Verfügung gestellt:

Die folgenden Methoden könnten für Sie nützlich sein und ihre Implementierung kann auch als Illustration dienen, wie die Klasse `TuringMachine` zu verwenden ist:

- Die Methode `TuringMachine.idle` erzeugt eine TM, die nichts tut (also die Identitätsfunktion berechnet).
- Die Methode `m1.addMachine(m2)` fügt alle Zustände, Endzustände und Übergänge aus der TM `m2` zu `m1` hinzu.
- Die Methode `m1.compose(m2)` macht aus der TM `m1` die Hintereinanderausführung von `m1` und `m2`.
- Die Methode `WhileToTM.deleteTape` demonstriert in etwa das Muster, wie die von Ihnen zu implementierenden Methoden aussehen soll. Sie erzeugt eine TM, die ein bestimmtes Band komplett löscht.

Zur Notation: Das “Blank”-Zeichen  $\square$  aus der Vorlesung wird als Leerzeichen ' ' geschrieben. Außerdem gibt es ein besonderes Wildcard-Zeichen  $*$ :

- Ein Übergang  $*/X, R$  bedeutet “egal was unter dem Lesekopf steht, schreibe ein  $X$  und bewege den Kopf nach rechts”.
- Ein Übergang  $X/*, R$  bedeutet “wenn ein  $X$  unter dem Lesekopf steht, lasse es stehen und bewege den Kopf nach rechts”.
- Ein Übergang  $*/*, R$  bedeutet “egal was unter dem Lesekopf steht, lasse es stehen und bewege den Kopf nach rechts”.
- Die Konstanten `ANYTHING` und `NO_CHANGE` sind Abkürzungen für  $\underbrace{** \dots *}_{n+1 \text{ mal}}$ , sodass Sie leicht Übergänge schreiben können, die auf jeden Bandinhalt matchen bzw. die den Bandinhalt nicht verändern.
- Analog ist `DONT_MOVE` eine Abkürzung für  $\underbrace{NN \dots N}_{n+1 \text{ mal}}$ .
- Die Methode `makeLetters` nimmt eine Funktion  $f : \mathbb{N} \rightarrow \text{char}$  und gibt den Vektor  $(f(0), \dots, f(n))$  zurück. Damit können Sie z.B. den Vektor, der aus einem  $X$  an Position  $i$  und sonst nur aus  $*$  besteht schreiben als `makeLetters(j -> (j == i) ? 'X' : '*')`.

Ihre konkrete Aufgabe ist nun wie folgt: Implementieren Sie die diversen `visit`-Methoden des `Visitors` `WhileToTM`. Es bietet sich an, dafür zuerst die ebenfalls mit `TODO` markierten Hilfsfunktionen `copyTape`, `incrementTape` und `decrementTape` zu implementieren. Das Bandalphabet sowie die zu verwendende Anzahl an Bändern ist dabei bereits im Kontext des `Visitors` vorhanden.

## Bewertungshinweise

Um zu testen, ob Sie eine korrekte TM zurückgeben, wird Ihre TM für verschiedene Eingaben simuliert und geprüft, ob das Ergebnis mit dem des ursprünglichen `WHILE`-Programms übereinstimmt. Um Nichttermination zu vermeiden, darf Ihre TM dabei nur eine bestimmte, testfallspezifische Anzahl an Schritten machen, bevor sie terminiert (“fuel”). Diese Zahl wurde großzügig gewählt, sodass Ihnen mit einer halbwegs sinnvollen Konstruktion nicht das “fuel” ausgehen sollte. Sollte dies dennoch passieren, so terminiert Ihre TM entweder nicht oder Ihre Konstruktion ist deutlich zu kompliziert.

Auf TUMjudge gibt es zwei “Aufgaben” zu dieser Aufgabe. Der Code, den Sie hochladen müssen, ist beide Male der gleiche; der erste Test testet jedoch nur einzelne `Increment/Decrement`-Anweisungen ( $x_i := x_j + c$  und  $x_i := x_j - c$ ) und Blöcke. Damit ist es Ihnen möglich, bereits für die korrekte Implementierung von `Increment/Decrement` und `Block` einen Punkt zu bekommen.

## Eingabe

Die Eingabe ist ein `WHILE`-Programm in der oben definierten Syntax. Zusätzlich werden davor zwei Zahlen `Max fuel` und `Variable range` angegeben, die für Sie nicht relevant sind.

## Ausgabe

Eine zum gegebenen `WHILE`-Programm äquivalente TM wie oben beschrieben. Die Syntax hierfür ist:

- eine Zeile mit dem Alphabet
- eine Zeile mit dem Startzustand
- eine Zeile mit den Endzuständen
- eine Zeile mit der Anzahl der Bänder

- beliebig viele Zeilen mit Transitionen. Eine Transition  $q \xrightarrow{ab/cd, LR} q'$  wird dabei geschrieben als `q; ab; q' ; cd; LR`.
- eine Zeile mit dem Inhalt `END`

## Beispiele

Weitere Beispiele finden Sie in dem .tar.gz-Archiv auf der Vorlesungswebseite.

### Sample Input 1

```
Max fuel: 100
Variable range: 5
x0 := x0 + 0
EOF
```

### Sample Output 1

```
X
1
1
1
END
```

### Sample Input 2

```
Max fuel: 9500
Variable range: 5
x1 := x0 + 0
EOF
```

### Sample Output 2

```
X
7
11
2
10; *; 11; **; RR
10; X*; 10; *X; LL
7; * ; 8; **; NN
7; *X; 7; * ; NR
8; **; 9; **; NN
9; *; 10; **; LN
9; X*; 9; **; RN
END
```

### Sample Input 3

```
Max fuel: 1000
Variable range: 5
x0 := x0 + 1
EOF
```

### Sample Output 3

```
X
15
17
1
15; *; 16; *; L
16; *; 17; X; N
END
```

### Sample Input 4

```
Max fuel: 500
Variable range: 5
x0 := x0 - 1
EOF
```

### Sample Output 4

```
X
20
21
1
20; *; 21; ; R
END
```

### Sample Input 5

```
Max fuel: 100
Variable range: 5
x0 := x0 - 0
EOF
```

### Sample Output 5

```
X
23
23
1
END
```

**Sample Input 6**

```
Max fuel: 1500
Variable range: 5
x0 := x0 + 2
EOF
```

**Sample Output 6**

```
X
28
31
1
28;*;29;*;L
29;*;30;X;L
30;*;31;X;N
END
```

**Sample Input 7**

```
Max fuel: 1000
Variable range: 5
x0 := x0 - 2
EOF
```

**Sample Output 7**

```
X
35
37
1
35;*;36; ;R
36;*;37; ;R
END
```

**Sample Input 8**

```
Max fuel: 2000
Variable range: 5
x0 := x0 + 3
EOF
```

**Sample Output 8**

```
X
43
47
1
43;*;44;*;L
44;*;45;X;L
45;*;46;X;L
46;*;47;X;N
END
```

**Sample Input 9**

```
Max fuel: 1500
Variable range: 5
IF x0 = 0 THEN
  x1 := x1 + 1
ELSE
  x1 := x1 - 1
END
EOF
```

**Sample Output 9**

```
X
54
57;59
2
54;*;55;*;NN
54;X*;58;*;NN
55;*;56;*;NL
56;*;57;*X;NN
58;*;59;* ;NR
END
```