Script generated by TTT

Title: Seidl: Theoretische_Informatik (17.06.2013)

Date: Mon Jun 17 10:20:27 CEST 2013

Duration: 87:10 min

Pages: 68

Definition 4.10
Eine Turingmaschine M akzeptiert die Sprache

$$L(M) = \{w \in \Sigma^* \mid \exists q \in F, \alpha, \beta \in \Gamma^*.\ (\epsilon, q_0, w) \rightarrow_M^* (\alpha, q, \beta)\}$$

Definition 4.10
Eine Turingmaschine M akzeptiert die Sprache

$$L(M) = \{w \in \Sigma^* \mid \exists q \in F, \alpha, \beta \in \Gamma^*.\ (\epsilon, q_0, w) \rightarrow_M^* (\alpha, q, \beta)\}$$

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt Turing-berechenbar gdw es eine Turingmaschine M gibt, so dass für alle $n_1, \ldots n_k, m \in \mathbb{N}$ gilt

$$f(n_1, \ldots, n_k) = m \quad \Leftrightarrow$$
$$\exists r \in F.\ (\epsilon, q_0, bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k)) \rightarrow_M^* (\square\ldots\square, r, bin(m)\square\ldots\square)$$

wobei $bin(n)$ die Binärdarstellung der Zahl n ist.

Definition 4.10
Eine Turingmaschine M akzeptiert die Sprache

$$L(M) = \{w \in \Sigma^* \mid \exists q \in F, \alpha, \beta \in \Gamma^*.\ (\epsilon, q_0, w) \rightarrow_M^* (\alpha, q, \beta)\}$$

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt Turing-berechenbar gdw es eine Turingmaschine M gibt, so dass für alle $n_1, \ldots n_k, m \in \mathbb{N}$ gilt

$$f(n_1, \ldots, n_k) = m \quad \Leftrightarrow$$
$$\exists r \in F.\ (\epsilon, q_0, bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k)) \rightarrow_M^* (\square\ldots\square, r, bin(m)\square\ldots\square)$$

wobei $bin(n)$ die Binärdarstellung der Zahl n ist.

Eine Funktion $f : \Sigma^* \rightarrow \Sigma^*$ heißt Turing-berechenbar gdw es eine Turingmaschine M gibt, so dass für alle $u, v \in \Sigma^*$ gilt

$$f(u) = v \quad \Leftrightarrow \quad \exists r \in F.\ (\epsilon, q_0, u) \rightarrow_M^* (\square\ldots\square, r, v\square\ldots\square)$$

Zum Halten/Terminieren von TM

OE(!) können wir festlegen, dass eine TM in der Konfiguration (α, q, β) hält falls $q \in F$.

Zum Halten/Terminieren von TM

OE(!) können wir festlegen, dass eine TM in der Konfiguration (α, q, β) hält falls $q \in F$. Dann gibt es keine Konfiguration (α', q', β') mit $(\alpha, q, \beta) \to_M (\alpha', q', \beta')$.

Ist δ partiell, so kann eine TM auch halten, bevor sie einen Endzustand erreicht.

Zum Halten/Terminieren von TM

OE(!) können wir festlegen, dass eine TM in der Konfiguration (α, q, β) hält falls $q \in F$. Dann gibt es keine Konfiguration (α', q', β') mit $(\alpha, q, \beta) \to_M (\alpha', q', \beta')$.

Ist δ partiell, so kann eine TM auch halten, bevor sie einen Endzustand erreicht.

Daher können wir auch annehmen, dass $\delta(q, a)$ undefiniert (bzw $\emptyset$) ist falls $q \in F$.

Satz 4.11
Zu jeder nichtdeterministischen TM N gibt es eine deterministische TM M mit $L(N) = L(M)$.

Beweis:
M durchsucht den Baum der Berechnungen von N in *Breitensuche*, beginnend mit der Startkonfiguration (ϵ, q_0, w), eine Ebene nach der anderen.

Satz 4.11

Zu jeder nichtdeterministischen TM N gibt es eine deterministische TM M mit $L(N) = L(M)$.

Beweis:

M durchsucht den Baum der Berechnungen von N in *Breitensuche*, beginnend mit der Startkonfiguration (ϵ, q_0, w), eine Ebene nach der anderen.
Gibt es in dem Baum (auf Ebene n) eine Konfigurationen mit Endzustand, so wird diese (nach Zeit $O(c^n)$) gefunden. □

Satz 4.12

Die von Turingmaschinen akzeptierten Sprachen sind genau die Typ-0-Sprachen der Chomsky Hierarchie.

Satz 4.12

Die von Turingmaschinen akzeptierten Sprachen sind genau die Typ-0-Sprachen der Chomsky Hierarchie.

Beweis:

Wir beschreiben nur die Beweisidee. (Mehr Details: [Schöning]).

Satz 4.12

Die von Turingmaschinen akzeptierten Sprachen sind genau die Typ-0-Sprachen der Chomsky Hierarchie.

Beweis:

Wir beschreiben nur die Beweisidee. (Mehr Details: [Schöning]).

„$\Longrightarrow$“: Grammatikregeln können direkt die Rechenregeln einer TM simulieren.

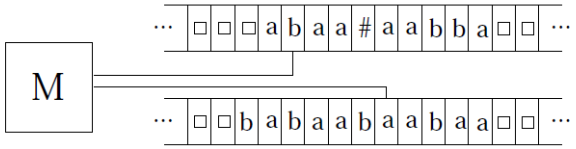
Satz 4.12
Die von Turingmaschinen akzeptierten Sprachen sind genau die Typ-0-Sprachen der Chomsky Hierarchie.

Beweis:
Wir beschreiben nur die Beweisidee. (Mehr Details: [Schöning]).

„$\Longrightarrow$“: Grammatikregeln können direkt die Rechenregeln einer TM simulieren.

„$\Longleftarrow$“: Die (nichtdeterministische) TM versucht von ihrer Eingabe aus das Startsymbol der Grammatik zu erreichen, indem sie die Produktionen der Grammatik von rechts nach links anwendet, (nichtdeterministisch) an jeder möglichen Stelle. □

Eine beliebte Modellvariante ist die k-Band-Turingmaschine:



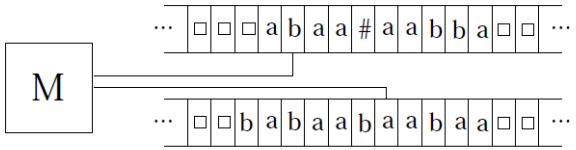
Die k Köpfe sind völlig unabhängig voneinander:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^k$$

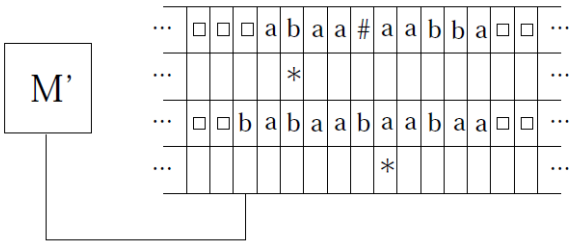
Satz 4.13
Jede k-Band-Turingmaschine kann effektiv durch eine 1-Band-TM simuliert werden.

Satz 4.13
Jede k-Band-Turingmaschine kann effektiv durch eine 1-Band-TM simuliert werden.

Beweisidee: Aus



wird



Beweisskizze:

- $\Gamma' := (\Gamma \times \{\star, \square\})^k$

Beweisskizze:

- $\Gamma' := (\Gamma \times \{\star, \square\})^k$
- M' simuliert einen M-Schritt durch mehrere Schritte: M'
  - startet mit Kopf links von allen $\star$.
  - geht nach rechts bis alle $\star$ überschritten sind, und merkt sich dabei (in Q') die Zeichen über jedem $\star$.
  - hat jetzt alle Information, um δ_M anzuwenden.
  - geht nach links über alle $\star$ hinweg und führt dabei δ_M aus.

Beobachtung:

n Schritte von M lassens sich durch
$O(n^2)$ Schritte von M' simulieren.

Denn nach n Schritten von M trennen $\leq 2n$ Felder linkesten und rechtesten Kopf.

4.3 Programmieren von Mehrband-TM

Die folgenden Basismaschinen sind leicht programmierbar:

- Band i := Band $i + 1$
- Band i := Band $i - 1$
- Band i := 0
- Band i := Band j

Seien $M_i = (Q_i, \Sigma, \Gamma_i, \delta_i, q_i, \square, F_i)$, $i = 1, 2$.
Die sequentielle Komposition (Hintereinanderschaltung) von M_1 und M_2 bezeichnen wir mit

$$\longrightarrow M_1 \longrightarrow M_2 \longrightarrow$$

Seien $M_i = (Q_i, \Sigma, \Gamma_i, \delta_i, q_i, \square, F_i)$, $i = 1, 2$.
Die sequentielle Komposition (Hintereinanderschaltung) von M_1 und M_2 bezeichnen wir mit

$$\longrightarrow M_1 \longrightarrow M_2 \longrightarrow$$

Sie ist wie folgt definiert:

$$M := (Q_1 \cup Q_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, q_1, \square, F_2)$$

wobei (oE) $Q_1 \cap Q_2 = \emptyset$

Seien $M_i = (Q_i, \Sigma, \Gamma_i, \delta_i, q_i, \square, F_i)$, $i = 1, 2$.
Die sequentielle Komposition (Hintereinanderschaltung) von M_1 und M_2 bezeichnen wir mit

$$\longrightarrow M_1 \longrightarrow M_2 \longrightarrow$$

Sie ist wie folgt definiert:

$$M := (Q_1 \cup Q_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, q_1, \square, F_2)$$

wobei (oE) $Q_1 \cap Q_2 = \emptyset$ und

$$\delta := \delta_1 \cup \delta_2 \cup \{(f_1, a) \mapsto (q_2, a, N) \mid f_1 \in F_1, a \in \Gamma_1\}$$

Sind f_1 und f_2 Endzustände von M so bezeichnet

$$\begin{array}{lccc} \longrightarrow & M & \xrightarrow{f_1} & M_1 \longrightarrow \\ & \downarrow f_2 & & \\ & M_2 & & \\ & \downarrow & & \end{array}$$

eine Fallunterscheidung,

Seien $M_i = (Q_i, \Sigma, \Gamma_i, \delta_i, q_i, \square, F_i)$, $i = 1, 2$.
Die sequentielle Komposition (Hintereinanderschaltung) von M_1 und M_2 bezeichnen wir mit

$$\longrightarrow M_1 \longrightarrow M_2 \longrightarrow$$

Sie ist wie folgt definiert:

$$M := (Q_1 \cup Q_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, q_1, \square, F_2)$$

wobei (oE) $Q_1 \cap Q_2 = \emptyset$ und

$$\delta := \delta_1 \cup \delta_2 \cup \{(f_1, a) \mapsto (q_2, a, N) \mid f_1 \in F_1, a \in \Gamma_1\}$$

Sind f_1 und f_2 Endzustände von M so bezeichnet

$$\begin{array}{lcccl} \longrightarrow & M & \xrightarrow{f_1} & M_1 & \longrightarrow \\ & \downarrow f_2 & & & \\ & M_2 & & & \\ & \downarrow & & & \end{array}$$

eine Fallunterscheidung,

Sind f_1 und f_2 Endzustände von M so bezeichnet

$$\begin{array}{lcccl} \longrightarrow & M & \xrightarrow{f_1} & M_1 & \longrightarrow \\ & \downarrow f_2 & & & \\ & M_2 & & & \\ & \downarrow & & & \end{array}$$

eine Fallunterscheidung, dh eine TM, die vom Endzustand f_1 von M nach M_1 übergeht, und von f_2 aus nach M_2.

Die folgende TM nennen wir „Band=0?“ (bzw „Band $i = 0$?“):

Sind f_1 und f_2 Endzustände von M so bezeichnet

$$\begin{array}{lcccl} \longrightarrow & M & \xrightarrow{f_1} & M_1 & \longrightarrow \\ & \downarrow f_2 & & & \\ & M_2 & & & \\ & \downarrow & & & \end{array}$$

eine Fallunterscheidung, dh eine TM, die vom Endzustand f_1 von M nach M_1 übergeht, und von f_2 aus nach M_2.

Die folgende TM nennen wir „Band=0?“ (bzw „Band $i = 0$?“):

$$\begin{array}{rcl} \delta(q_0, 0) & = & (q_0, 0, R) \\ \delta(q_0, \Box) & = & (ja, \Box, L) \end{array}$$

Sind f_1 und f_2 Endzustände von M so bezeichnet

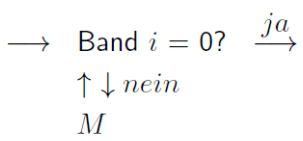
$$\begin{array}{lcccl} \longrightarrow & M & \xrightarrow{f_1} & M_1 & \longrightarrow \\ & \downarrow f_2 & & & \\ & M_2 & & & \\ & \downarrow & & & \end{array}$$

eine Fallunterscheidung, dh eine TM, die vom Endzustand f_1 von M nach M_1 übergeht, und von f_2 aus nach M_2.

Die folgende TM nennen wir „Band=0?“ (bzw „Band $i = 0$?“):

$$\begin{array}{rcll} \delta(q_0, 0) & = & (q_0, 0, R) & \\ \delta(q_0, \Box) & = & (ja, \Box, L) & \\ \delta(q_0, a) & = & (nein, a, N) & \text{für } a \neq 0, \Box \end{array}$$

Analog zur Fallunterscheidung kann man auch eine TM für eine Schleife konstruieren



Analog zur Fallunterscheidung kann man auch eine TM für eine Schleife konstruieren

die sich wie `while` Band $i \neq 0$ `do` M verhält.

Analog zur Fallunterscheidung kann man auch eine TM für eine Schleife konstruieren

die sich wie `while` Band $i \neq 0$ `do` M verhält.

Moral: Mit TM kann man imperativ programmieren:

`:=`
`;`
`if`
`while`

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

Syntax von LOOP-Programmen:

$$
\begin{aligned}
P \;\to\; & X \texttt{ := } X \texttt{ + } C \\
\mid\; & X \texttt{ := } X \texttt{ - } C \\
\mid\; & P\texttt{; } P \\
\mid\; & \texttt{IF } X = 0 \texttt{ DO } P \texttt{ ELSE} Q \texttt{ END} \\
\mid\; & \texttt{LOOP } X \texttt{ DO } P \texttt{ END}
\end{aligned}
$$

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

Syntax von LOOP-Programmen:

$$
\begin{aligned}
P \;\to\; & X \texttt{ := } X \texttt{ + } C \\
\mid\; & X \texttt{ := } X \texttt{ - } C \\
\mid\; & P\texttt{; } P \\
\mid\; & \texttt{IF } X = 0 \texttt{ DO } P \texttt{ ELSE} Q \texttt{ END} \\
\mid\; & \texttt{LOOP } X \texttt{ DO } P \texttt{ END}
\end{aligned}
$$

wobei X eine der Variablen $x_0, x_1, \ldots$
und C eine der Konstanten $0, 1, \ldots$ sein kann.

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

Syntax von LOOP-Programmen:

$$
\begin{aligned}
P \;\to\; & X \texttt{ := } X \texttt{ + } C \\
\mid\; & X \texttt{ := } X \texttt{ - } C \\
\mid\; & P\texttt{; } P \\
\mid\; & \texttt{IF } X = 0 \texttt{ DO } P \texttt{ ELSE} Q \texttt{ END} \\
\mid\; & \texttt{LOOP } X \texttt{ DO } P \texttt{ END}
\end{aligned}
$$

wobei X eine der Variablen $x_0, x_1, \ldots$
und C eine der Konstanten $0, 1, \ldots$ sein kann.

Beispiel 4.14

`LOOP` x_2 `DO` x_1 `:=` x_0`+1` `END`

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i `:=` x_j `+` n Neuer Wert von x_i ist $x_j + n$.

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i `:=` x_j `+` n Neuer Wert von x_i ist $x_j + n$.

x_i `:=` x_j `-` n Neuer Wert von x_i ist $x_j \dot{-} n$.

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i `:=` x_j `+` n Neuer Wert von x_i ist $x_j + n$.

x_i `:=` x_j `-` n Neuer Wert von x_i ist $x_j \dot{-} n$.

P_1 `;` P_2 Führe zuerst P_1 und dann P_2 aus.

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m-n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i := x_j + n Neuer Wert von x_i ist $x_j + n$.

x_i := x_j - n Neuer Wert von x_i ist $x_j \dot{-} n$.

P_1 ; P_2 Führe zuerst P_1 und dann P_2 aus.

LOOP x_i DO P END Führe P genau n mal aus, wobei n der *Anfangswert* von x_i ist.

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m-n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i := x_j + n Neuer Wert von x_i ist $x_j + n$.

x_i := x_j - n Neuer Wert von x_i ist $x_j \dot{-} n$.

P_1 ; P_2 Führe zuerst P_1 und dann P_2 aus.

LOOP x_i DO P END Führe P genau n mal aus, wobei n der *Anfangswert* von x_i ist. Zuweisungen an x_i in P ändern die Anzahl n der Schleifendurchläufe nicht.

Beispiel 4.15

LOOP x_0 DO x_1 := x_1+1 END simuliert x_1 := x_1 + x_0

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m-n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i := x_j + n Neuer Wert von x_i ist $x_j + n$.

x_i := x_j - n Neuer Wert von x_i ist $x_j \dot{-} n$.

P_1 ; P_2 Führe zuerst P_1 und dann P_2 aus.

LOOP x_i DO P END Führe P genau n mal aus, wobei n der *Anfangswert* von x_i ist. Zuweisungen an x_i in P ändern die Anzahl n der Schleifendurchläufe nicht.

Beispiel 4.15

LOOP x_0 DO x_1 := x_1+1 END simuliert x_1 := x_1 + x_0

Zu Beginn der Ausführung stehen die Eingaben in $x_1, \ldots, x_k$.
Alle anderen Variablen sind 0. Die Ausgabe wird in x_0 berechnet.

Definition 4.16

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Lemma 4.17
Alle LOOP-berechenbaren Funktionen sind total.

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Lemma 4.17
Alle LOOP-berechenbaren Funktionen sind total.

Beweis mit Induktion über die Erzeugung/Struktur der LOOP-Programme.

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Lemma 4.17
Alle LOOP-berechenbaren Funktionen sind total.

Beweis mit Induktion über die Erzeugung/Struktur der LOOP-Programme.

Sind alle totalen Funktionen LOOP-berechenbar?

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j
  (falls $i \neq j, k$)
- DIV, MOD, ...
- x := komplexer Ausdruck

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j
  (falls $i \neq j, k$)
- DIV, MOD, ...
- x := komplexer Ausdruck
- IF $x = 0$ THEN P END

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j
  (falls $i \neq j, k$)
- DIV, MOD, ...
- x := komplexer Ausdruck
- IF $x = 0$ THEN P END

WHILE-Programme sind LOOP-Programme erweitert um WHILE-Schleifen:

$$P \rightarrow \texttt{WHILE } X \neq 0 \texttt{ DO } P \texttt{ END}$$

WHILE-Programme sind LOOP-Programme erweitert um WHILE-Schleifen:

$$P \rightarrow \texttt{WHILE } X \neq 0 \texttt{ DO } P \texttt{ END}$$

Die Semantik ist wie üblich.

Fakt 4.18
WHILE-Schleifen können LOOP-Schleifen simulieren.

WHILE-Programme sind LOOP-Programme erweitert um WHILE-Schleifen:

$$P \rightarrow \texttt{WHILE } X \neq 0 \texttt{ DO } P \texttt{ END}$$

Die Semantik ist wie üblich.

Fakt 4.18
WHILE-Schleifen können LOOP-Schleifen simulieren.

Fakt 4.19
LOOP-Schleifen können WHILE-Schleifen nicht simulieren.

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:
P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:
P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)
- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:
P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)
- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,
- terminiert nicht, falls $f(n_1, \ldots, n_k)$ undefiniert ist.

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,
- terminiert nicht, falls $f(n_1, \ldots, n_k)$ undefiniert ist.

Turingmaschinen können WHILE-Programme simulieren:

Satz 4.21 (WHILE → TM)
Jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,
- terminiert nicht, falls $f(n_1, \ldots, n_k)$ undefiniert ist.

Turingmaschinen können WHILE-Programme simulieren:

Satz 4.21 (WHILE → TM)
Jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

Beweis:
Jede Programmvariable wird auf einem eigenen Band gespeichert.

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

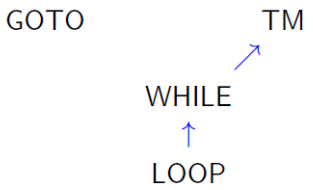
P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,
- terminiert nicht, falls $f(n_1, \ldots, n_k)$ undefiniert ist.

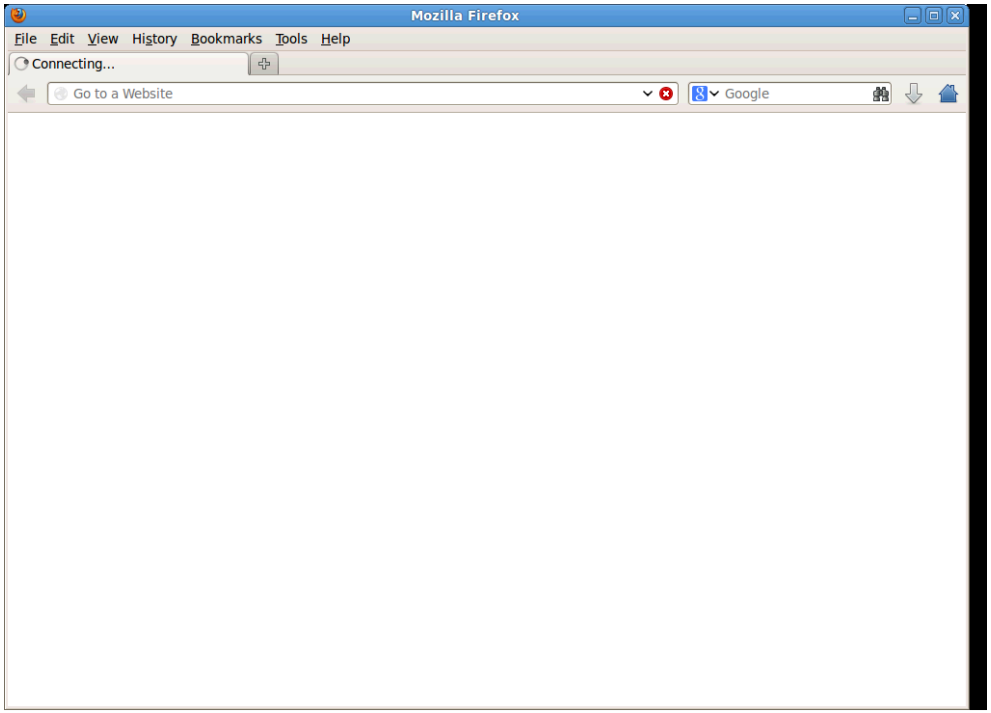
Turingmaschinen können WHILE-Programme simulieren:

Satz 4.21 (WHILE → TM)
Jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

Beweis:
Jede Programmvariable wird auf einem eigenen Band gespeichert.
Wir haben bereits gezeigt: Alle Konstrukte der WHILE-Sprache können von einer Mehrband-TM simuliert werden,



Übersetzungen


Mozilla Firefox
File Edit View History Bookmarks Tools Help
Connecting...
Go to a Website
Google