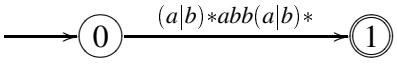
Universität des Saarlandes
Fachrichtung 6.2 – Informatik
Prof. Dr. R. Wilhelm
Dr.-Ing. D. Kästner
Dipl.-Inf. M. Langenbach

10. Mai 2001

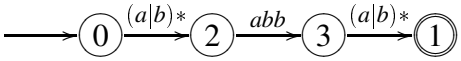
Lösungsvorschlag zum Übungsblatt 2: Übersetzerbau (SS 2001)

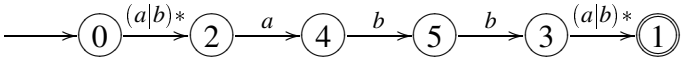
Aufgabe 1 (8 Punkte)

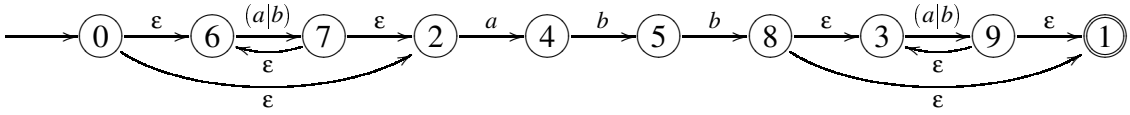
Zunächst ist aus dem regulären Ausdruck ein nichtdeterministischer endlicher Automat zu konstruieren. Dabei wird mit einem Automaten begonnen, der nur aus einem Startzustand, einem Endzustand und einer diese beiden Zustände verbindenden Kante besteht, die mit dem regulären Ausdruck beschriftet ist:

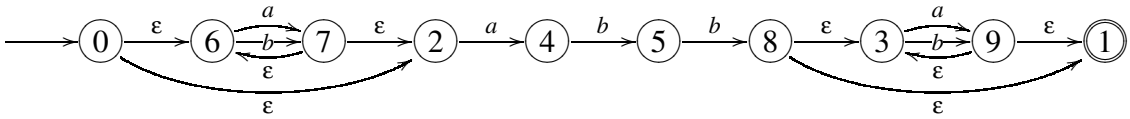


Davon ausgehend wird gemäß des Algorithmus *RA* $\rightarrow$ *NEA* der Automat immer mehr verfeinert, bis nur noch Symbole aus $\Sigma \cup \varepsilon$ als Kantenmarkierungen vorkommen.









Um aus dem nichtdeterministischen einen deterministischen endlichen Automaten zu konstruieren, findet der Algorithmus *NEA* $\rightarrow$ *DEA* Verwendung. Dieser wird initialisiert mit:

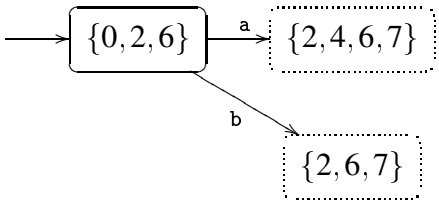
$$\begin{aligned} q_0' &= \{0,2,6\} \\ Q' &= \{\{0,2,6\}\} \\ \text{marked}(q_0') &= \textbf{false} \\ \delta &= \emptyset \end{aligned}$$

Im folgenden wird mit T_x die Menge von Zuständen bezeichnet, die im Algorithmus bei Betrachtung des Zeichens $x \in \Sigma$ in T enthalten sind.

Der erste betrachtete Zustand in Q', der noch nicht markiert wurde, ist $S = \{0,2,6\}$. Betrachtet man die Folgezustände unter den Zeichen `a`, und `b` abgeschlossen unter ε, dann erhält man:

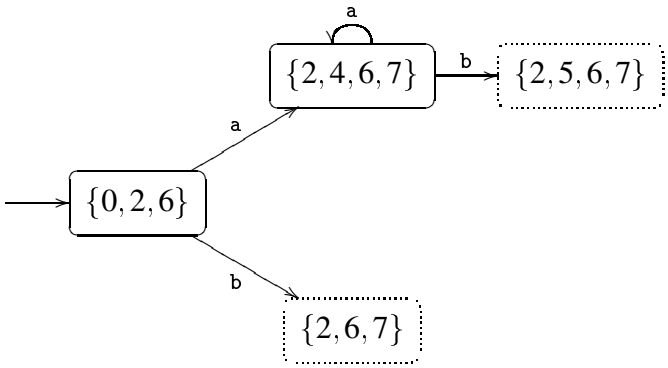
$$\begin{aligned} T_{\mathtt{a}} &= \{2,4,6,7\} \\ T_{\mathtt{b}} &= \{2,6,7\} \end{aligned}$$

$T_{\mathtt{a}}$ und $T_{\mathtt{b}}$ sind noch nicht in Q' enthalten, werden der Zustandsmenge also hinzugefügt, desweiteren auch Kanten von S nach $T_{\mathtt{a}}$ bzw. $T_{\mathtt{b}}$ unter dem Symbol `a` bzw. `b`. Der bis dahin erzeugte Graph hat dann folgendes Aussehen (noch nicht markierte Zustände sind mit einer gepunkteten Linie umgeben):

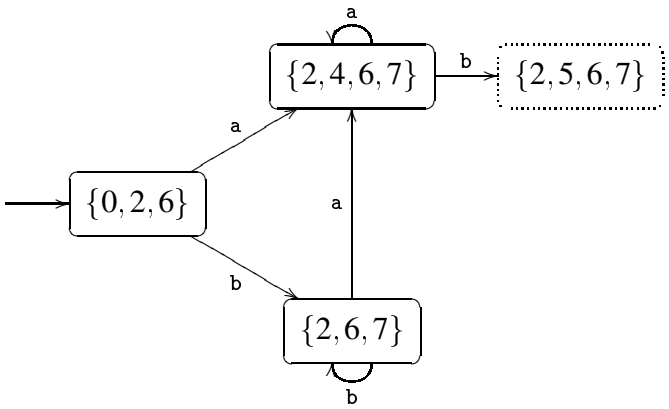


Analog wird für die verbleibenden nichtmarkierten Zustände verfahren.

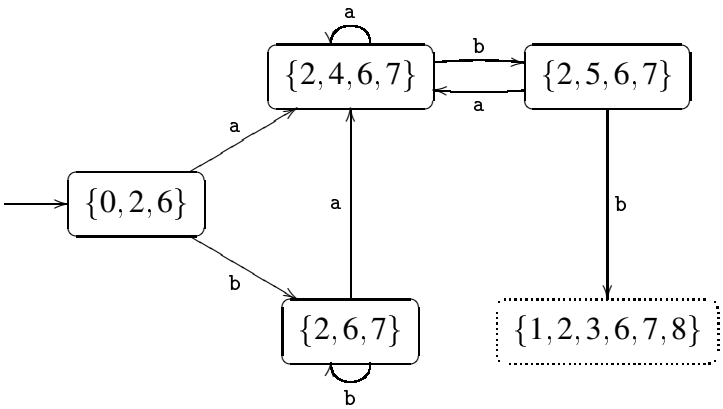
$$\begin{aligned} S &= \{2,4,6,7\} \\ T_{\mathtt{a}} &= \{2,4,6,7\} \\ T_{\mathtt{b}} &= \{2,5,6,7\} \end{aligned}$$



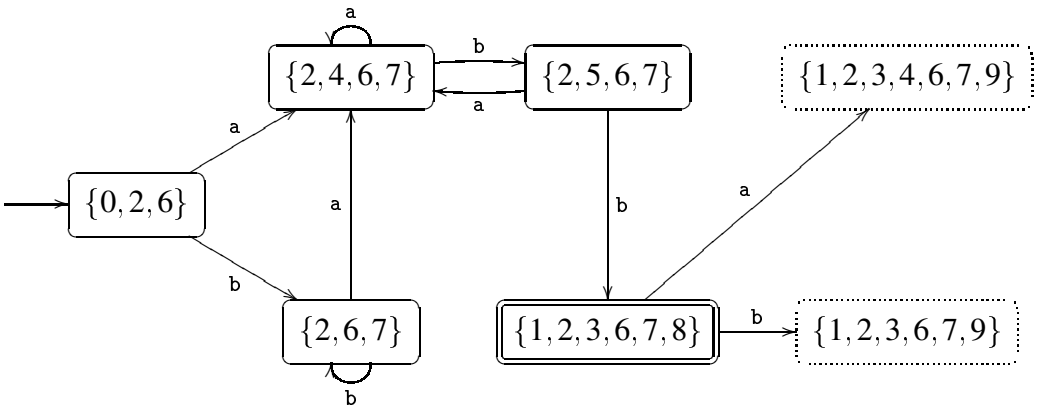
$$
\begin{aligned}
S &= \{2,6,7\} \\
T_{\mathtt{a}} &= \{2,4,6,7\} \\
T_{\mathtt{b}} &= \{2,6,7\}
\end{aligned}
$$



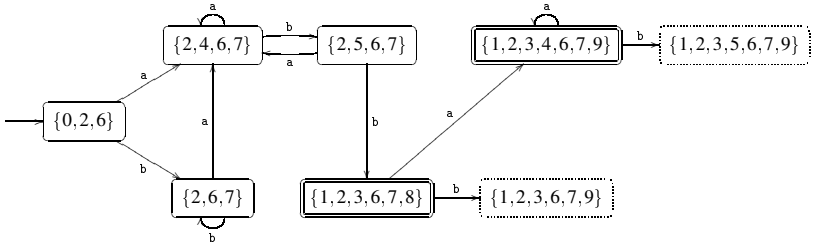
$$
\begin{aligned}
S &= \{2,5,6,7\} \\
T_{\mathtt{a}} &= \{2,4,6,7\} \\
T_{\mathtt{b}} &= \{1,2,3,6,7,8\}
\end{aligned}
$$



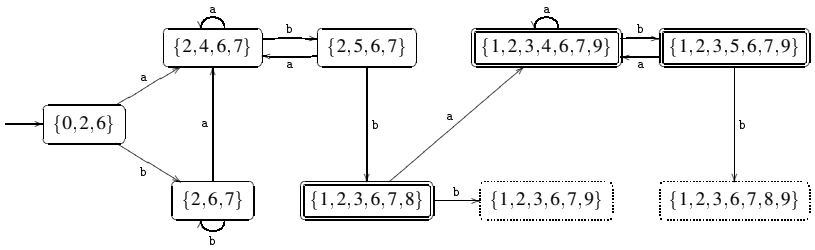
$$
\begin{aligned}
S &= \{1,2,3,6,7,8\} \\
T_{\mathtt{a}} &= \{1,2,3,4,6,7,9\} \\
T_{\mathtt{b}} &= \{1,2,3,6,7,9\}
\end{aligned}
$$



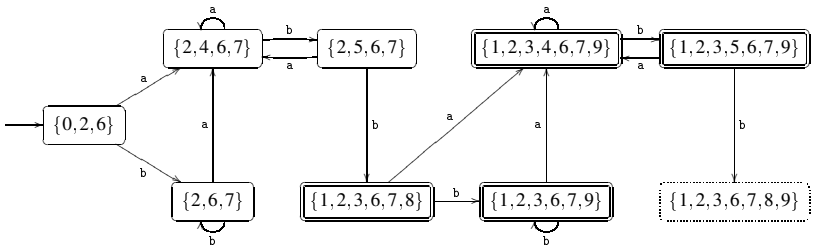
$$
\begin{aligned}
S &= \{1,2,3,4,6,7,9\} \\
T_{\mathtt{a}} &= \{1,2,3,4,6,7,9\} \\
T_{\mathtt{b}} &= \{1,2,3,5,7,8,9\}
\end{aligned}
$$



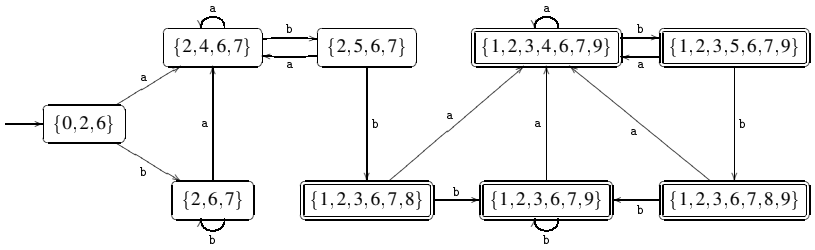
$$
\begin{aligned}
S &= \{1,2,3,5,6,7,9\} \\
T_{\mathtt{a}} &= \{1,2,3,4,6,7,9\} \\
T_{\mathtt{b}} &= \{1,2,3,6,7,8,9\}
\end{aligned}
$$



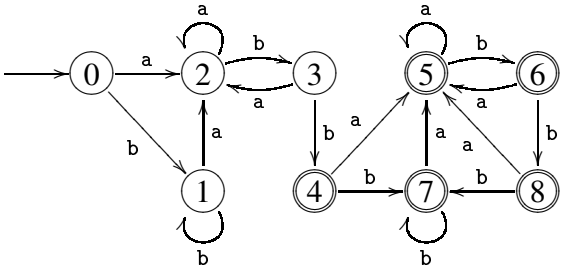
$$
\begin{aligned}
S &= \{1,2,3,6,7,9\} \\
T_{\mathsf{a}} &= \{1,2,3,4,7,9\} \\
T_{\mathsf{b}} &= \{1,2,3,6,7,9\}
\end{aligned}
$$



$$
\begin{aligned}
S &= \{1,2,3,6,7,8,9\} \\
T_{\mathsf{a}} &= \{1,2,3,4,6,7,9\} \\
T_{\mathsf{b}} &= \{1,2,3,6,7,9\}
\end{aligned}
$$



Das gleiche Übergangsdiagramm mit neuer Knotenmarkierung sieht dann so aus:



Um diesen Automaten zu minimieren, verwenden wir den Algorithmus *minDEA*. Während der Ausführungszeit wird eine Partition unterhalten, die Mengen von Zuständen beinhaltet, die durch ein Wort $w \in \Sigma^*$ *getrennt* werden. Ein Wort w trennt einen Zustand p von einem Zustand q, wenn der deterministische endliche Automat M ausgehend vom Zustand p durch Abarbeitung von w in einen akzeptierenden Zustand gelangt, aber ausgehend von q bei gleicher Eingabe w in einen nichtakzeptierenden Zustand kommt oder umgekehrt.

Gestartet wird das Verfahren mit $\Pi = \{Q-F, F\} = \{\{0,1,2,3\},\{4,5,6,7,8\}\}$. In diesem Fall ist das die Zustände trennende Wort ε.

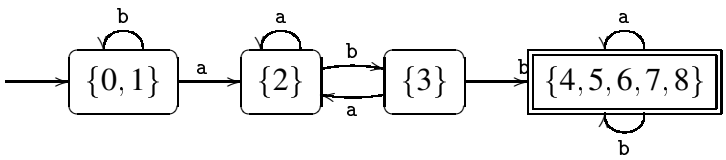
Betrachten wir nun die Menge $\{4,5,6,7,8\}$. Diese Menge kann nicht weiter verfeinert werden, da alle möglichen Teilpartitionen für jedes Symbol des Alphabets das gleiche Übergangsverhalten bezüglich der Ursprungspartition besitzen.

Die Menge $\{0,1,2,3\}$ läßt sich in die Teilpartitionen $\{0,1,2\}$ und $\{3\}$ unterteilen. Betrachtet man Übergänge von Zuständen aus der ersten Menge unter jedem beliebigen Symbol, ist der Folgezustand wieder in der ursprünglichen Partitionsmenge $\{0,1,2,3\}$. Vom Zustand 3 aus gibt es aber einen Übergang, nämlich unter dem Zeichen `b`, der diese Menge verläßt.

Damit erhlaten wir die verfeinerte Partition $\Pi = \{\{0,1,2\},\{3\},\{4,5,6,7,8\}\}$. Die Partition hat sich verändert, der Algorithmus fährt fort.

In einem weiteren Schritt wird $\{0,1,2\}$ in $\{0,1\}$ und $\{2\}$ unterteilt,da unter `b` ein Übergang von 2 aus existiert, der $\{0,1,2\}$ verläßt. Sämtliche Folgezustände von $\{0,1\}$ sind wieder in dieser Menge enthalten.

Das Ergebnis ist die Partition $\Pi = \{\{0\},\{1,2\},\{3\},\{4,5,6,7,8\}\}$. die sich nicht mehr aufspalten läßt. Wenn die Zustände, die sich in der gleichen Menge befinden, zusammengefaßt werden, erhält man den folgenden minimalen deterministischen endlichen Automaten, der den regulären Ausdruck erkennt:



Aufgabe 2 (8 Punkte)

Zunächst die Codesequenz:

iconst_0
istore 1

iconst_5
newarray `I`
astore 2

iconst_1
istore 1
aload 2
iconst_1
iconst_0
iastore

Für jede Methode muß die maximale Anzahl der benötigten Stackzellen berechnet werden. Da sowohl für die Initialisierung der lokalen Variablen als auch für die sich daran anschließenden Anweisungen u. U. Stackzellen benötigt werden, ist das Maximum aus Deklarationen und Anweisungen zu bilden.

stackNeed(`[static]` *methname* `(`arglist`)` { *decls statements* })
= max(*stackNeed*(*decls*), *stackNeed*(*statements*))

Nach jeder Anweisung (oder Deklaration) hat der Stack wieder seine ursprüngliche Tiefe. Damit läßt sich die Funktion *stackNeed* auf Anweisungsfolgen wie folgt definieren:

stackNeed(*statement*`;` *statement_list*)
= max(*stackNeed*(*statement*), *stackNeed*(*statement_list*))

Deklarationen mit Initialisierung werten die rechte Seite aus und sichern das Ergebnis. Bei der Deklaration von Objektvariablen ist zu beachten, daß die Objektreferenz zuvor auf den Stack gelegt wurde.

stackNeed(*type varname*=*expr*`;`)

$$= \begin{cases} 1+\textit{stackNeed}(\textit{expr}) & \text{falls } \textit{varname} \text{ Objektvariable} \\ \textit{stackNeed}(\textit{expr}) & \text{sonst} \end{cases}$$

Betrachten wir zunächst zusammengesetze Anweisungen: while- und do-while-Schleifen führen eine Anweisungsfolge aus und testen einen Ausdruck. Der Platzbedarf ist das Maximum.

stackNeed(`while (`*expr*`)` *statements*)
= max(*stackNeed*(*expr*), *stackNeed*(*statements*))

stackNeed(`do` *statements* `while (`*expr*`)`)
= max(*stackNeed*(*expr*), *stackNeed*(*statements*))

Analog geht man bei for-Schleifen, if-then-else-Konstrukten und switch-Anweisungen vor:

stackNeed(`for (`*statement*$_1$`;` *expr*`;` *statement*$_2$`)` *statements*)
= max(*stackNeed*(*expr*), *stackNeed*(*statement*$_1$),
stackNeed(*statement*$_2$), *stackNeed*(*statements*))

stackNeed(`if (`*expr*`)` *statements*)
= max(*stackNeed*(*expr*), *stackNeed*(*statements*))

stackNeed(`if (`*expr*`)` *statements*$_1$ `else` *statements*$_2$)
= max(*stackNeed*(*expr*), *stackNeed*(*statements*$_1$), *stackNeed*(*statements*$_2$))

stackNeed(`switch (`*expr*`)` *caselist*)
= max(*stackNeed*(*expr*), *stackNeed*(*caselist*))

stackNeed(`case` *number*`:` *statements caselist*)
= max(*stackNeed*(`case`) *number: statements*, *stackNeed*(*caselist*))

stackNeed(`case` *number*`:` *statements*)
= max(*stackNeed*(*statements*))

Unäre Operatoren manipulieren üblicherweise nur die oberste belegte Kellerzelle. Da es in der JVM keine gesonderte Instruktion zur Negation boolescher Werte gibt, muß man sich mit einer exklusiven Verorderung behelfen. Da die Negation von Wahrheitswerten allerdings nur in Verbindung mit Vergleich auftritt (in *TassKaf* gibt es keine bolleschen Variablen), läßt sich die Berechnung des Platzbedarfs für unäre Operatoren generisch halten:

stackNeed(*unop expr*)
= max(*stackNeed*(*expr*))

Bei binären Operatoren wird zunächst der erste Ausdruck ausgewertet. Dessen Ergebnis steht dann auf dem Stack, während sich der zweite Ausdruck in der Auswertung befindet:

stackNeed(*expr*$_1$ *binop expr*$_2$)
= max(*stackNeed*(*expr*$_1$), 1+*stackNeed*(*expr*$_2$))

Bei return-Anweisungen muß der Rückgabewert berechnet werden:

stackNeed(`return` *expr*)
= max(*stackNeed*(*expr*))

Die angegebenen Dimensionen bei new-Anweisungen werden sukzessive auf den Stack gelegt, so daß man direkt die Hilfsfunktion *stackNeedExpressionList* verwenden kann:

$$\begin{aligned} &\mathit{stackNeed}(\texttt{new}\ \mathit{type}\,\texttt{[}\mathit{expr}_1\texttt{]}\ \ldots\ \texttt{[}\mathit{expr}_k\texttt{]}) \\ &\quad = \mathit{stackNeedExpressionList}(\mathit{expr}_1, \ldots, \mathit{expr}_k)\ 0 \end{aligned}$$

Bei Zuweisungen ergibt die Interpretation der linken Seite i. d. R. eine Referenz, die dann zum Abspeichern der rechten Seite verwendet wird. Bei Arrays werden aber zur Ablage des Datums zwei Werte benötigt, nämlich Basisreferenz und Offset. Daraus ergibt sich:

$$\begin{aligned} &\mathit{stackNeed}(\mathit{expr}_1{=}\mathit{expr}_2) \\ &\quad = \max(\mathit{stackNeed}(\mathit{expr}_1), \mathit{inc}{+}\mathit{stackNeed}(\mathit{expr}_2)) \\ &\qquad \text{wobei } \mathit{inc} = \begin{cases} 2 & \text{falls } \mathit{expr}_1 \text{ eine Feldreferenz ist} \\ 1 & \text{sonst} \end{cases} \end{aligned}$$

Vor einem Methodenaufruf wird zunächst der Präfix ausgewertet. Danach werden die einzelnen Ausdrücke abgearbeitet, an die die formalen Parameter gebunden werden. Bei nicht-statischen Methoden befindet sich vor Auswertung der Parameter eine Objektreferenz auf dem Stack.

$$\begin{aligned} &\mathit{stackNeed}(\mathit{callee}\,\texttt{.}\,\mathit{methname}\ \texttt{(}\mathit{expr}_1, \ldots, \mathit{expr}_k\texttt{)}) \\ &\quad = \max(\mathit{stackNeed}(\mathit{callee}), \mathit{inc}{+}\mathit{stackNeedExpressionList}(\mathit{expr}_1, \ldots, \mathit{expr}_k)\ 0) \\ &\qquad \text{wobei } \mathit{inc} = \begin{cases} 0 & \text{falls } \mathit{methname} \text{ statisch} \\ 1 & \text{sonst} \end{cases} \end{aligned}$$

Bei der Berechnung des Platzbedarfs der Ausdrucksliste ist zu beachten, daß nach Auswertung der ersten n Ausdrücke bereits n Zellen belegt werden. Damit ergibt sich folgende rekursive Funktion:

$$\mathit{stackNeedExpressionList}()\ k = 0$$

$$\begin{aligned} &\mathit{stackNeedExpressionList}(\mathit{expr, rest})\ k \\ &\quad = \max(k{+}\mathit{stackNeed}(\text{expr}), \mathit{stackNeedExpressionList}(\text{rest})\ k+1) \end{aligned}$$

Bei Variablen muß man zunächst zwischen den verschiedenen Zugriffsarten unterscheiden. Bei statischen und lokalen Variablen kann direkt auf die Speicherzelle zugegriffen werden, Objektvariablen erfordern das vorherige Laden der Objektreferenz:

$$\mathit{stackNeed}(x) = \begin{cases} 1 & \text{falls statisch} \\ 1 & \text{falls local} \\ 2 & \text{sonst} \end{cases}$$

Feldinidzierungen sind mit der Ablage von aktuellen Parametern auf den Stack zu vergleichen. Beim Auswerten der n-ten Dimension befinden sich bereits $n-1$ Werte auf dem Stack.

stackNeed(*x*`[`*expr*$_1$`]` ... `[`*expr*$_k$`]`)
= max(*stackNeed*(*x*), 1+*stackNeedExpressionList*(*expr*$_1$, ..., *expr*$_k$) 0)

Bei Präfixen ist zu beachten, daß diese nur bei nichtstatischen Variablen benötigt werden.

stackNeed(*callee*`.`*x*)

$$= \begin{cases} \textit{stackNeed}(x) & \text{falls } x \text{ statisch} \\ \max(\textit{stackNeed}(\textit{callee}), 1+\textit{stackNeed}(x)) & \text{sonst} \end{cases}$$

Angewandt auf die Methode *foo()* ergibt sich, daß über die beiden Deklarationen – die im folgenden als Zuweisungen behandelt werden – und den Zuweisungen das Maximum zu bilden ist.

stackNeed(`x=0`)
= max(*stackNeed*(`x`), 1+*stackNeed*(`0`))
= max(1, 1+1)
= 2

stackNeed(`y=new int[5]`)
= max(*stackNeed*(`y`), 1+*stackNeed*(`new int[5]`))
= max(1, 1+*stackNeedExpressionList*(`5`) 0)
= max(1, 1+1)
= 2

stackNeed(`x=1`)
= 2

stackNeed(`y[1]=0`)
= max(*stackNeed*(`y[1]`), 2+*stackNeed*(`0`))
= max(max(*stackNeed*(`y`), 1+*stackNeedExpressionList*(`1`) 0, 2+1)
= max(max(1, 1+1), 3)
= 3

Damit ergibt sich ein Platzbedarf von 3 Stackzellen.