6.5.5 Analyse der verfügbaren Ausdrücke (Available Expressions)

Ein Ausdruck `b op c` ist an einem Punkt p im Programm verfügbar, wenn auf jedem Weg von Eingangsknoten *entry* zum Punkt p dieser Ausdruck ausgewertet wird und danach auf dem Weg nach p keine Zuweisungen zu `b` oder `c` passieren. Diese Information ist wichtig, um gemeinsame Teilausdrücke über Blockgrenzen hinaus zu erkennen.
Sei U die Menge aller Ausdrücke, die in Drei-Adress-Befehlen des Programms auftreten.
Zunächst bestimmt man wie in den bisherigen Fällen die Transferfunktion für einen Drei-Adress-Befehl s der Form `a := b op c`. Offensichtlich wird an dieser Stelle der Ausdruck `b op c` erzeugt und jeder Ausdruck aus U, der `a` enthält, gelöscht. Für den Spezialfall, dass `a` = `b` oder `a` = `c` gilt, wird der Ausdruck `b op c` also sofort wieder gelöscht!
Also hätten wir $\texttt{e\_gen}[s] = \{\texttt{b op c}\}$ und $\texttt{e_kill}[s] = \{$Ausdrücke aus U, die `a` enthalten$\}$ und die Transferfunktion wäre $f_s(x) = (\texttt{e\_gen}[s] \cup x) - \texttt{e\_kill}[s]$.

Analog folgt für andere Formen von Drei-Adress-Befehlen:

Befehlstyp s	`e_gen`$[s]$	`e_kill`$[s]$
`a := b+c`	{`b+c`}	alle Ausdrücke aus U, die `a` enthalten
`a := b[c]`	{`b[c]`}	alle Ausdrücke aus U, die `a` enthalten
`if a relOp b goto ...`	{}	{}
`a[b] := c`	{}	Ausdrücke aus U der Form `a[x]`
usw.		

Wie üblich kann man jetzt aus den Transferfunktionen für die einzelnen Befehle $s_1, \ldots, s_k$ eines Blocks eine Transferfunktion für den gesamten Block konstruieren.
Man erhält

$$f_B(x) = \texttt{e\_gen}[B] \cup (x - \texttt{e\_kill}[B])$$

wobei

$$\texttt{kill}[B] = \texttt{e\_kill}[1] \cup \texttt{e\_kill}[2] \cup \ldots \cup \texttt{e\_kill}[k]$$

und

$$\begin{aligned} \texttt{e\_gen}[B] \quad = \quad & (\texttt{e\_gen}[k] - \texttt{e\_kill}[k]) \cup (\texttt{gen}[k-1] - \texttt{e\_kill}[k-1] - \texttt{e\_kill}[k]) \cup \\ & \ldots \cup (\texttt{e\_gen}[1] - \texttt{e\_kill}[1] - \texttt{e\_kill}[2] - \cdots - \texttt{e\_kill}[k]) \end{aligned}$$

Algorithmus zur Bestimmung der `e_gen`- und `e_kill`-Menge eines Blocks

Eingabe: Ein einfacher Block `B`
Ausgabe: Die `e_gen` und `e_kill`-Menge für den Block `B`
Verfahren :

1) Setze `e_gen[B]` := $\emptyset$ und `e_kill[B]` := $\emptyset$.

2) Durchlaufe die Instruktionen des Blocks `B` vom ersten bis zum letzten Befehl und führe für jeden Befehl s die folgenden Schritte nacheinander aus:

   i) `e_gen[B]` := (`e_gen[B]` $\cup$ `e_gen`$[s]$) $-$ `e_kill`$[s]$

   ii) `e_kill[B]` := `e_kill[B]` $\cup$ `e_kill`$[s]$

Beispiel 6.14:
Man betrachte die den folgenden, aus 4 Befehlen bestehenden einfachen Block B. Es werden für jeden Programm-Punkt die verfügbaren Ausdrücke angegeben, wobei angenommen wird, dass vor Eintritt in diesen Block kein Ausdruck verfügbar ist.

Befehl	verfügbare Ausdrücke
	$\emptyset$
`a := b + c`	
	$\{$`b + c`$\}$
`b := a - d`	
	$\{$`a - d`$\}$
`c := b + c`	
	$\{$`a - d`$\}$
`d := a - d`	
	$\emptyset$

Es wäre also $\texttt{e\_gen}[B] = \emptyset$ und es gilt $\{\texttt{b + c}, \texttt{a - d}\} \subseteq \texttt{e\_kill}[B]$.

Offensichtlich haben wir hier eine Vorwärtsanalyse. Es gelten die Datenfluss-Gleichungen:

$$\texttt{out}[B] = \texttt{e\_gen}[B] \cup (\texttt{in}[B] - \texttt{e\_kill}[B])$$

$$\texttt{in}[B] = \bigcap_{P \in \text{pred}(B)} \texttt{out}[P]$$

Für den Eingangsblock „entry“ gilt offensichtlich die Randbedingung $\texttt{out}[\text{entry}] = \emptyset$.

Bemerkung: Man beachte, dass in der Bestimmungsgleichung für die `in`-Menge ein Durchschnittsoperator statt eines Vereinigungsoperators auftritt. Dies ist korrekt, da ein Ausdruck nur dann am Anfang eines Blocks verfügbar ist, wenn er am Ende *aller* Vorgängerblöcke verfügbar ist.

Die Lösung wird wieder iterativ bestimmt. Beim Start der Iteration muss man jedoch etwas anders vorgehen: Man setzt die `out`-Menge für den „entry“-Knoten auf die leere Menge und für alle anderen Knoten setzt man sie auf die Menge aller Ausdrücke U.

Algorithmus zur Bestimmung verfügbarer Ausdrücke

Eingabe: Ein Flussgraph, für dessen Blöcke die jeweiligen `e_gen`- und `e_kill`-Mengen bestimmt worden sind.
Ausgabe: Die `out` und `in`-Mengen für jeden Block im Flussgraphen.
Verfahren :
pred(`B`) sei die Menge der Vorgängerblöcke von `B` im Flussgraphen.

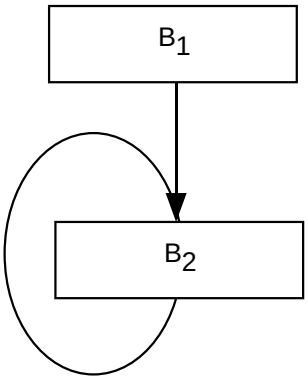
1) Setze `out`[B] := U, `out`[entry] = $\emptyset$.

2) Solange sich eine der `out`-Mengen ändert, führe man die folgenden Schritte für jeden Block `B` aus:

   i) `in[B]` := $\bigcap_{\texttt{P} \in \text{pred}(\texttt{B})}$ `out[P]`

   ii) `out[B]` : = `e_gen[B]` $\cup$ (`in[B]` - `e_kill[B]`)

Beispiel 6.15:
Sei folgendes Flussdiagramm gegeben:



Man betrachte den Block B_2. Man kann die zeitliche Entwicklung der `in`- und `out`-Werte durch die folgenden Rekursionen beschreiben, dabei seinen $\texttt{out}[B_2]_j$ bzw. $\texttt{in}[B_2]_j$ der Wert der `out`- bzw. `in`-Menge von $\texttt{B}_2$ zum Zeitpunkt j. Man erhält:

$$\begin{aligned} \texttt{in}[B_2]_{j+1} &= \texttt{out}[B_1]_j \cap \texttt{out}[B_2]_j \\ \texttt{out}[B_2]_{j+1} &= \texttt{e\_gen}[B_2] \cup (\texttt{in}[B_2]_{j+1} - \texttt{e\_kill}[B_2]) \end{aligned}$$

Würde man mit $\texttt{out}[B_2]_0 = \emptyset$ starten, dann wäre $\texttt{in}[B_2]_1 = \texttt{out}[B_1]_0 \cap \texttt{out}[B_2]_0 = \emptyset$. Das bedeutet aber, dass ein in B_1 generierter Ausdruck wie `a op b`, der in B_2 nicht auftritt und auch nicht in der Kill-Menge von B_2 vorkommt, nie in der `out`-Menge von B_2 auftaucht.
Startet man dagegen mit $\texttt{out}[B_2]_0 = U$, erhält man korrekterweise $\texttt{in}[B_2]_1 = \texttt{out}[B_1]_0 \cap \texttt{out}[B_2]_0 = \texttt{out}[B_1]$.

Hier zum Abschluss eine Tabelle mit den wichtigsten Parametern für die Datenfluss-Analyse:

	verfügbare Definitionen	Lebendigkeit	verfügbare Ausdrücke
Domain	Menge von Definitionen	Menge von Variablen	Menge von Ausdrücken
Richtung	Vorwärts	Rückwärts	Vorwärts
Transfer-Fkt.	$\texttt{gen}[B] \cup (x - \texttt{kill}[B])$	$\texttt{use}[B] \cup (x - \texttt{def}[B])$	$\texttt{e\_gen}[B] \cup (x - \texttt{e\_kill}[B])$
Gleichungen	$\texttt{out[B]} = f_B(\texttt{in[B]})$	$\texttt{in[B]} = f_B(\texttt{out[B]})$	$\texttt{out[B]} = f_B(\texttt{in[B]})$
	$\texttt{in[B]} = \cup_{B' \in \text{pred}[B]} \texttt{out[B']}$	$\texttt{out[B]} = \cup_{B' \in \text{succ}[B]} \texttt{in[B']}$	$\texttt{in[B]} = \cap_{B' \in \text{pred}[B]} \texttt{out[B']}$
Initialisierung	$\texttt{in}[B] = \texttt{out}[B] = \emptyset$	$\texttt{in}[B] = \texttt{out}[B] = \emptyset$	$\texttt{in}[B] = \texttt{out}[B] = N$
Randbedingung	$\texttt{out}[\text{entry}] = \emptyset$	$\texttt{in}[\text{exit}] = \emptyset$	$\texttt{out}[\text{entry}] = \emptyset$

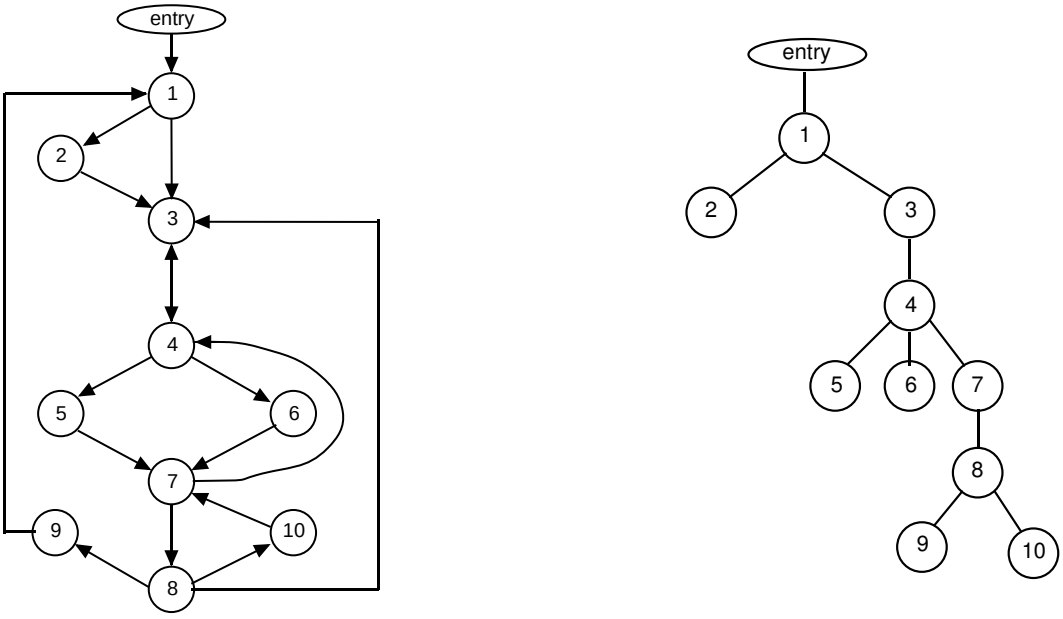
6.6 Analyse der Ablaufstruktur

Eine wichtige Aufgabe im Rahmen der Codeoptimierung ist das Erkennen von Schleifen, damit spätere Optimierungen speziell auf die Schleifenrümpfe angesetzt werden können.

Definition: Gegeben sei ein erweiterter Flussgraph. Block $\mathtt{B_d}$ **dominiert** Block `B`, geschrieben $\mathtt{B_d} \succeq \mathtt{B}$, falls $\mathtt{B_d}$ auf jedem Weg vom Knoten `entry` nach `B` liegt. Die Relation $\succeq$ ist offensichtlich eine Ordnung auf der Menge der Blöcke.
Block $\mathtt{B_1}$ ist **direkter Dominator-Block** vom Block $\mathtt{B_2}$, geschrieben $\mathtt{B_1} \succ \mathtt{B_2}$, falls für alle Blöcke `B` gilt: Aus $\mathtt{B} \succeq \mathtt{B_2}$ und $\mathtt{B} \neq \mathtt{B_2}$ folgt $\mathtt{B} \succeq \mathtt{B_1}$

Jeder Block außer `entry` hat einen eindeutigen direkten Dominator-Block.

Beispiel 6.16:
Gegeben sei der folgende erweiterte Flussgraph. Der rechts abgebildete Dominator-Baum gibt für jeden Block den direkten Dominator-Block als Vorgängerknoten an.



6.6.1 Algorithmus zur Berechnung der $\succeq$-Relation

Eingabe: Ein erweiterter Flussgraph mit der Knotenmenge N und einem Startknoten entry $\in N$.
Ausgabe: Die $\succeq$-Relation. Für jeden Knoten B aus N gibt $D(B)$ die Menge der Dominatoren von B an
Verfahren: Man löse die folgende Datenfluss-Analyse:

	Dominatoren
Domain	Potenzmenge von N
Richtung	Vorwärts
Transfer-Fkt.	$f_B(x) = x \cup \{B\}$
Gleichungen	$\mathtt{out[B]} = f_B(\mathtt{in[B]})$
	$\mathtt{in[B]} = \cap_{P \in \mathrm{pred}[B]} \mathtt{out[P]})$
Randbedingung	$\mathtt{out}[\mathrm{entry}] = \{\mathrm{entry}\}$
Initialisierung	$\mathtt{out[B]} = N$

Am Ende des Algorithmus gilt $\texttt{out}[B] = D(B)$ für alle Blöcke $B \in N$. Es ist $a \in D(b)$ genau dann wenn $a \succeq b$.

Beispiel 6.17:
Für den Flussgraphen aus Beispiel 6.16 ergibt sich im ersten Durchlauf, wenn man die Knoten in der Reihenfolge der Nummern betrachtet:

$$\begin{aligned}
D(1) &:= \{1\} \\
D(2) &:= \{2\} \cup D(1) = \{1,2\} \\
D(3) &:= \{3\} \cup (D(1) \cap D(2) \cap D(8)) = \{1,3\} \\
D(4) &:= \{4\} \cup (D(3) \cap D(7)) = \{1,3,4\} \\
D(5) &:= \{5\} \cup D(4) = \{1,3,4,5\} \\
D(6) &:= \{6\} \cup D(4) = \{1,3,4,6\} \\
D(7) &:= \{7\} \cup (D(5) \cap D(6) \cap D(10)) = \{1,3,4,7\} \\
D(8) &:= \{8\} \cup D(7) = \{1,3,4,7,8\} \\
D(9) &:= \{9\} \cup D(8) = \{1,3,4,7,8,9\} \\
D(10) &:= \{10\} \cup D(8) = \{1,3,4,7,8,10\}
\end{aligned}$$

Ein zweiter Durchlauf durch die Schleife bringt bei diesem Beispiel bereits keine weitere Veränderung. (Der Knoten *entry* wurde nicht aufgeführt, da er in allen D-Mengen enthalten ist.)

6.6.2 Natürliche Schleifen

Mit dieser Information ist nun möglich, Schleifen im Programmablauf zu erkennen. Als erstes überlegt man sich, welche Eigenschaften eine Schleife charakterisiert:

1) Eine Schleife hat einen einzigen **Eintrittspunkt** oder **Kopf** (Header). Dieser Knoten dominiert alle anderen Knoten in der Schleife.

2) Es muss mindestens einen Knoten in der Schleife geben, vom dem aus eine Kante zum Eintrittspunkt zurückführt.

Also muss man Kanten $a \to b$ im Flussgraphen suchen, für die $b \succeq a$ gilt, d.h. der Zielknoten b dominiert den Ausgangsknoten a.
Die Kanten nennt man **Rückwärtskanten** (Back-Edges). In unserem Beispiel 6.16 wären dies die Kanten $7 \to 4$, $10 \to 7$, $4 \to 3$, $8 \to 3$ und $9 \to 1$.

Definition: Die **natürliche Schleife** einer Rückwärtskante $n \to d$ mit $d \neq n$ besteht aus d und den Knoten, über die man n erreichen kann ohne über d zu gehen. Ist $n = d$, so besteht die natürliche Schleife nur aus dem Block d. d ist der Eintrittspunkt der Schleife.

Natürliche Schleifen haben die Eigenschaft, dass zwei Schleifen entweder disjunkt sind oder, sofern beide Schleifen nicht den gleichen Kopf besitzen, die eine in der anderen enthalten ist.
Für den Fall, dass zwei Schleifen den gleichen Kopf haben und keine in der anderen enthalten ist, scheint es sinnvoll zu sein, beide zusammen als eine Schleife aufzufassen.

Um die zu einer Rückwärtskante gehörende Schleife zu finden, kann man folgenden Algorithmus verwenden:

Algorithmus zur Konstruktion einer natürlichen Schleife

Eingabe: Ein Flussgraph und eine Rückwärtskante $n \to d$
Ausgabe: Eine Menge `loop`, die alle Knoten enthält, die in der natürlichen Schleife von $n \to d$ enthalten sind.
Verfahren :

```
stack := leer;
loop := {d};
if n = d return;
insert (n);
while stack ≠ leer do begin
    m := pop(stack);
    for jedem Vorgänger p von m do
        insert(p);
    end;

procedure insert(m)
if m ∉ loop then begin
    loop := loop ∪ {m}
    push (m, stack)
end
```

Beispiel 6.18:
Man betrachte wieder den Flussgraphen aus Beispiel 6.16.
Zur Kante $10 \to 7$ gehört die Schleife $\{7, 8, 10\}$.
Zur Kante $7 \to 4$ gehört die Schleife $\{4, 5, 6, 7, 8, 10\}$.
Die Kanten $4 \to 3$ und $8 \to 3$ definieren die Schleife $\{3, 4, 5, 6, 7, 8, 10\}$ und die Kante $9 \to 1$ definiert eine Schleife, die alle Knoten von 1 bis 10 enthält.
Die Knoten $\{4, 5, 6, 7\}$ bilden keine Schleife, da in diese Knotenmenge sowohl über 4 als auch über 7 eingetreten werden kann und damit die Kopf-Bedingung verletzt ist.