3 Ausnutzung von Hardware-Einrichtungen

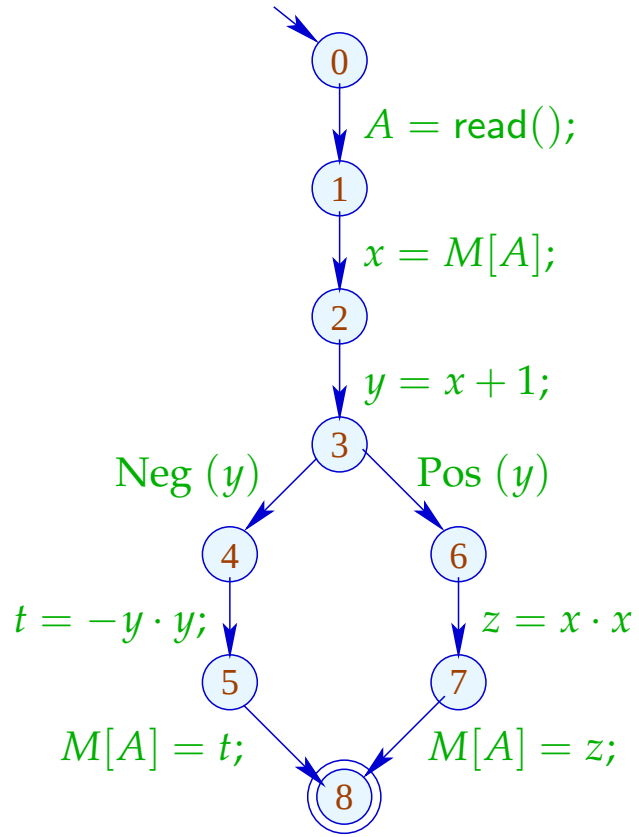
Frage: Wie nutzt man optimal

- ... Register
- ... Instruktionen
- ... Pipelines
- ... Caches
- ... Prozessoren ???

3.1 Register

Beispiel:

```
A = read();
x = M[A];
y = x + 1;
if (y) {
        z = x · x;
        M[A] = z;
} else {
        t = −y · y;
        M[A] = t;
}
```



Das Programm benötigt 5 Variablen ...

Problem:

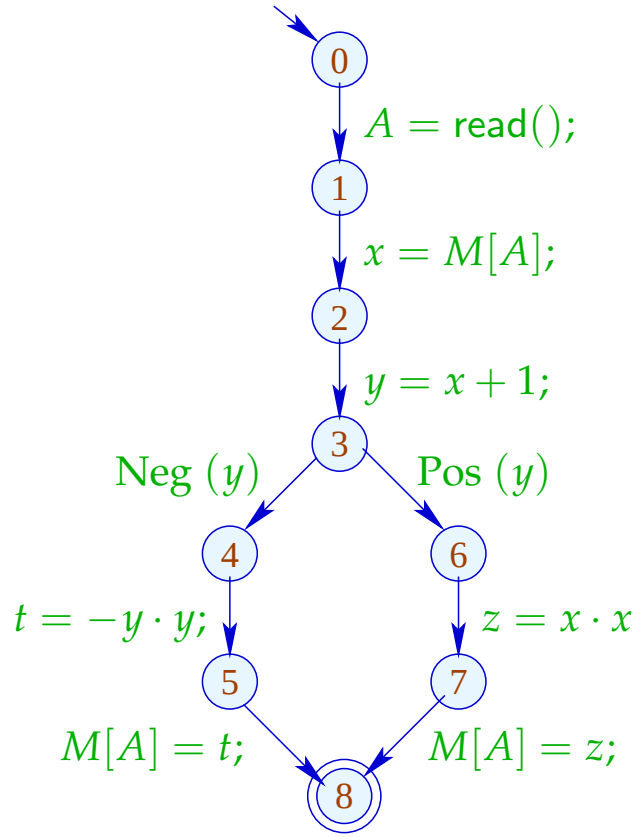
Was tun, wenn das Programm benutzt mehr Variablen als Register da sind :-(

Idee:

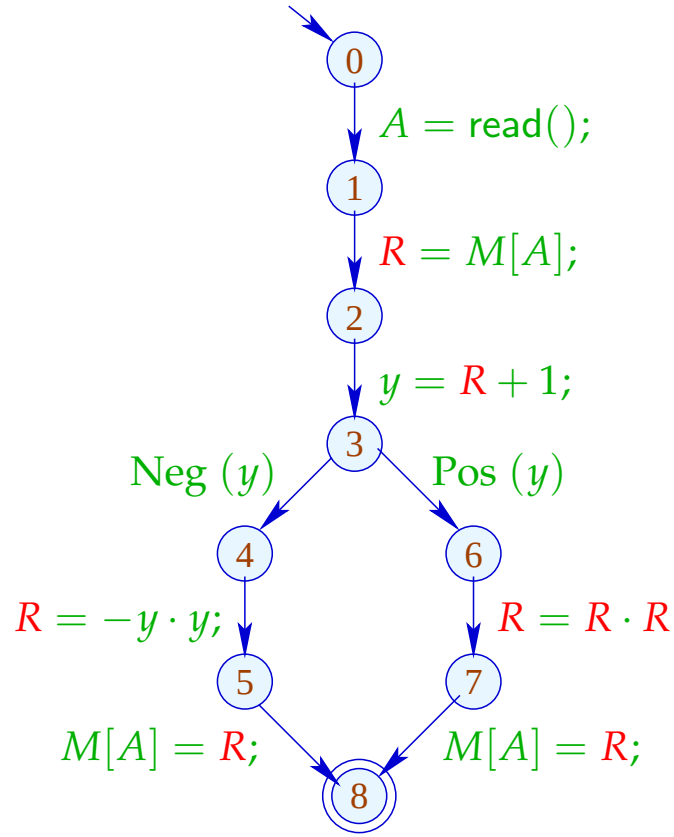
Benutze ein Register für mehrere Variablen :-)

Im Beispiel etwa eines für x, t, z ...

```
A = read();
x = M[A];
y = x + 1;
if (y) {
        z = x · x;
        M[A] = z;
} else {
        t = −y · y;
        M[A] = t;
}
```



```
A = read();
R = M[A];
y = R + 1;
if (y) {
        R = R · R;
        M[A] = R;
} else {
        R = −y · y;
        M[A] = R;
}
```



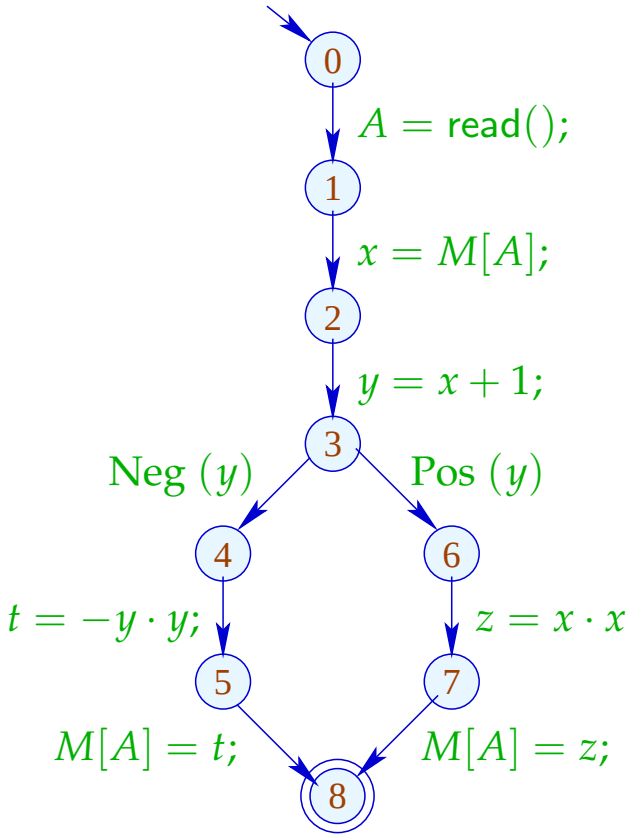
Achtung:

Das geht nur, wenn sich die Lebendigkeitsbereiche nicht überschneiden :-)

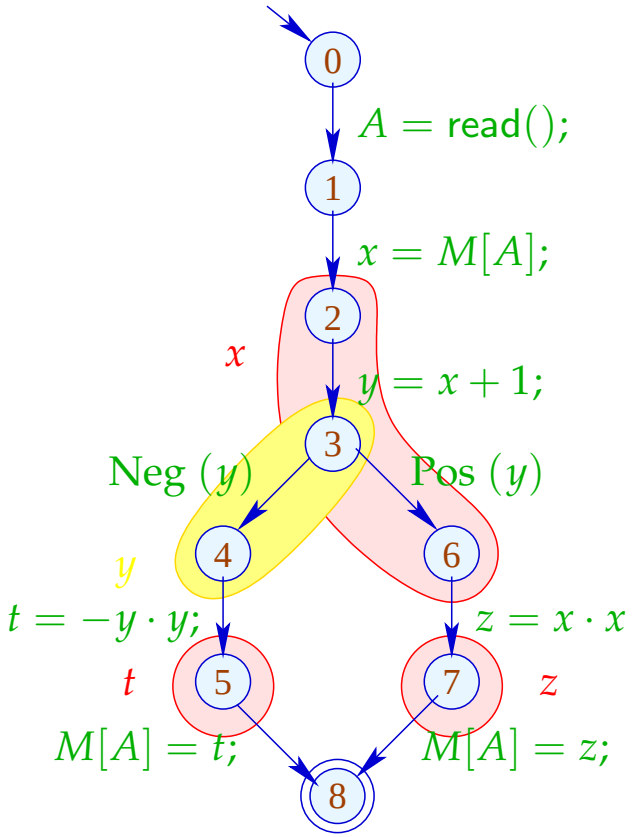
Der (wahre) Lebendigkeitsbereich von x ist:

$$\mathcal{L}[x] \;=\; \{u \mid x \in \mathcal{L}[u]\}$$

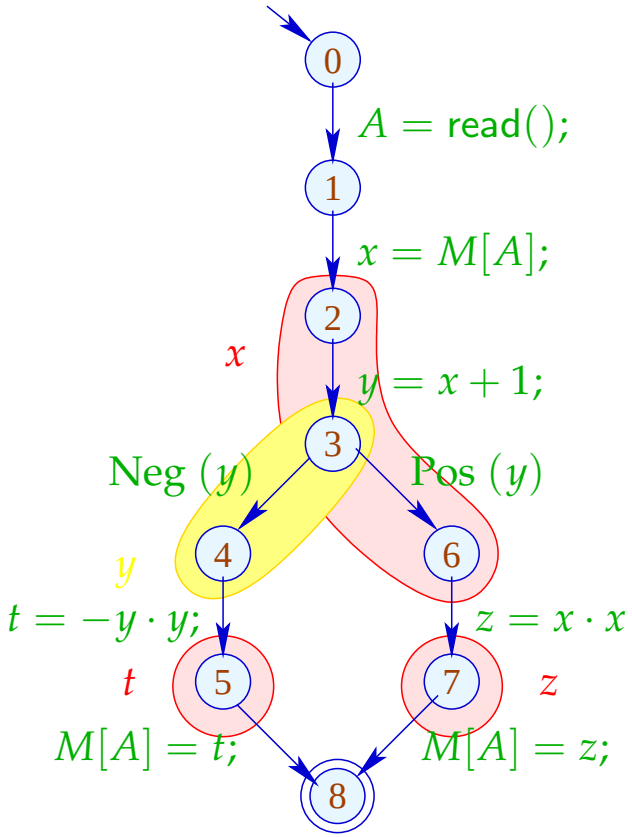
... im Beispiel:



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



Lebendigkeitsbereiche:

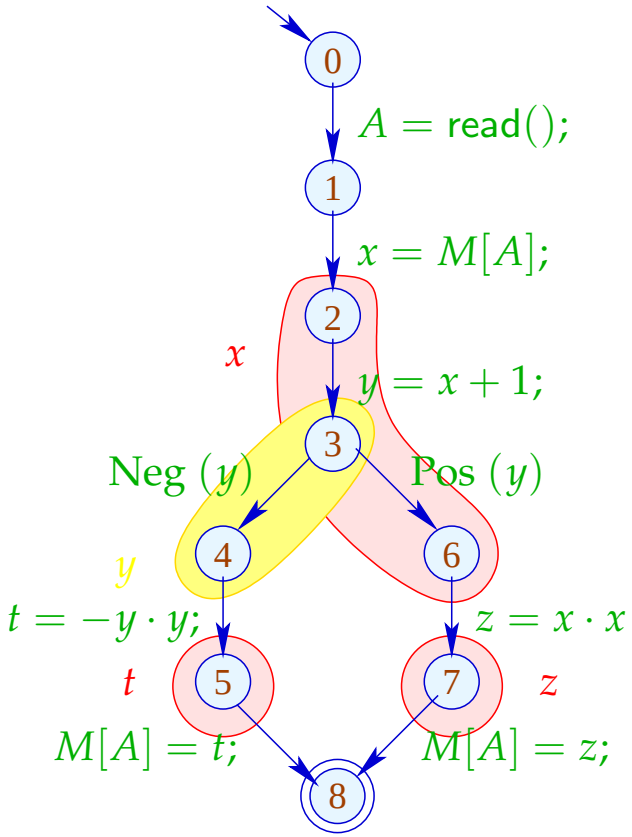
A	$\{1, \ldots, 7\}$
x	$\{2, 3, 6\}$
y	$\{2, 4\}$
t	$\{5\}$
z	$\{7\}$

Um Mengen kompatibler Variablen zu finden, konstruieren wir den Interferenz-Graphen $I = (Vars, E_I)$, wobei:

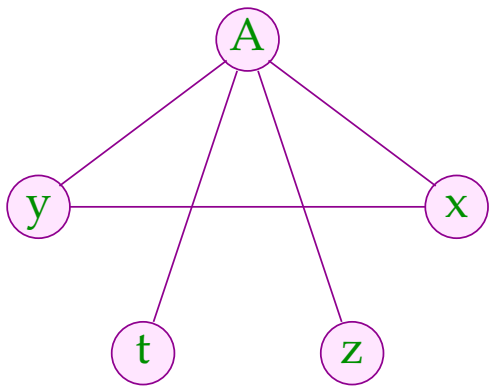
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

E_I enthält eine Kante für $x \neq y$ genau dann wenn x, y an einem gemeinsamen Punkt lebendig sind :-)

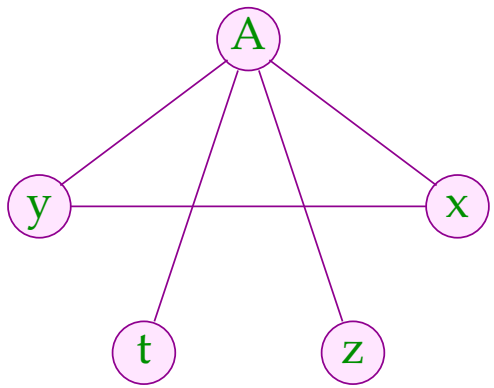
... im Beispiel:



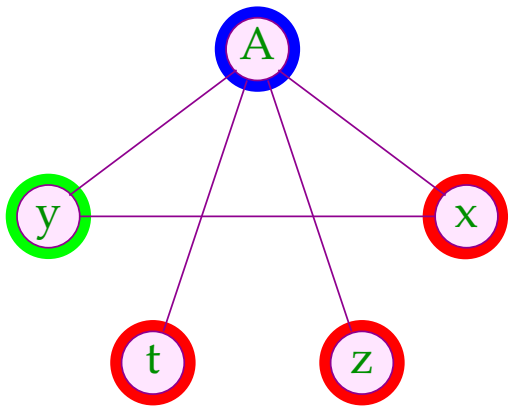
Interferenz-Graph:



Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)

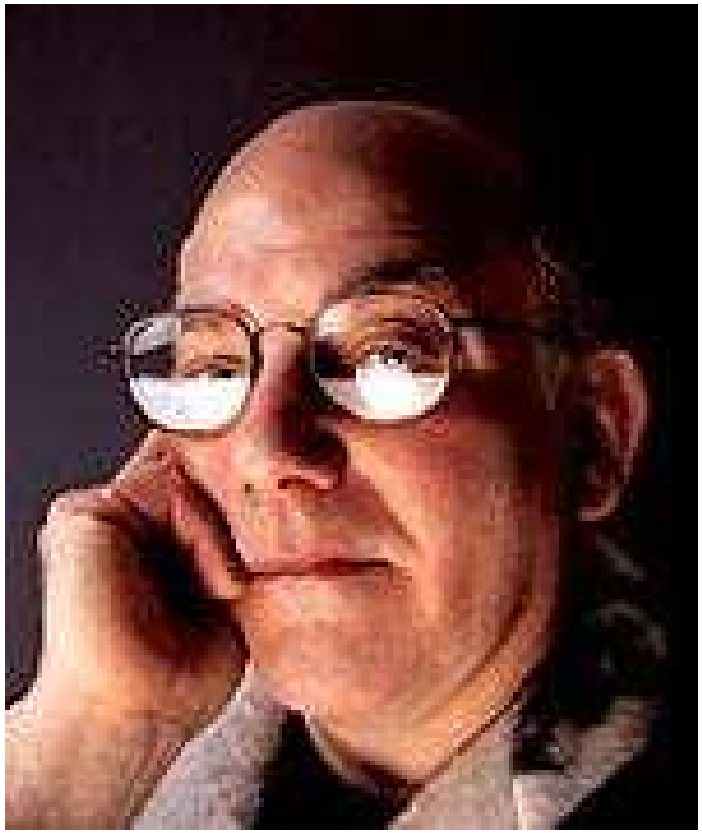


Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)



Farbe $\equiv$ Register

Sviatoslav Sergeevich Lavrov,
Russische Akademie der Wissenschaften (1962)



Gregory J. Chaitin, University of Maine (1981)

Abstraktes Problem:

Gegeben: Ungerichteter Graph (V, E) .

Gesucht: Minimale Färbung, d.h. Abbildung $c : V \to \mathbb{N}$ mit

(1) $c(u) \neq c(v)$ für $\{u, v\} \in E$;

(2) $\bigsqcup\{c(u) \mid u \in V\}$ minimal!

- Im Beispiel reichen 3 Farben :-) Aber Achtung:
- Die minimale Färbung ist i.a. nicht eindeutig :-(
- Es ist NP-vollständig herauszufinden, ob eine Färbung mit maximal k Farben möglich ist :-((

$\Longrightarrow$

Wir sind auf Heuristiken angewiesen oder Spezialfälle :-)

Greedy-Heuristik:

- Beginne irgendwo mit der Farbe 1;
- Wähle als jeweils neue Farbe die kleinste Farbe, die verschieden ist von allen bereits gefärbten Nachbarn;
- Ist ein Knoten gefärbt, färbe alle noch nicht gefärbten Nachbarn;
- Behandle eine Zusammenhangskomponente nach der andern

...

... etwas konkreter:

```
forall (v ∈ V) c[v] = 0;
forall (v ∈ V) color (v);

void color (v) {
        if (c[v] ≠ 0) return;
        neighbors = {u ∈ V | {u, v} ∈ E};
        c[v] = ⨅{k > 0 | ∀ u ∈ neighbors : k ≠ c(u)};
        forall (u ∈ neighbors)
                if (c(u) == 0) color (u);
}
```

Die neue Farbe lässt sich leicht berechnen, nachdem die Nachbarn nach ihrer Farbe geordnet wurden :-)

Diskussion:

- → Im wesentlichen ist das Prä-order DFS :-)
- → In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(
- → ... ist aber in der Praxis ganz gut :-)
- → ... Achtung: verschiedene Varianten sind patentiert !!!

Diskussion:

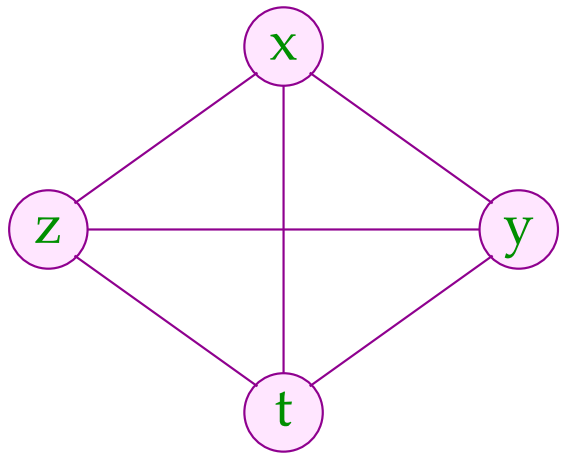
- → Im wesentlichen ist das Prä-order DFS :-)
- → In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(
- → ... ist aber in der Praxis ganz gut :-)
- → ... Achtung: verschiedene Varianten sind patentiert !!!

Der Algorithmus funktioniert umso besser, je kleiner die Lebendigkeitsbereiche sind ...

Idee: Life range splitting

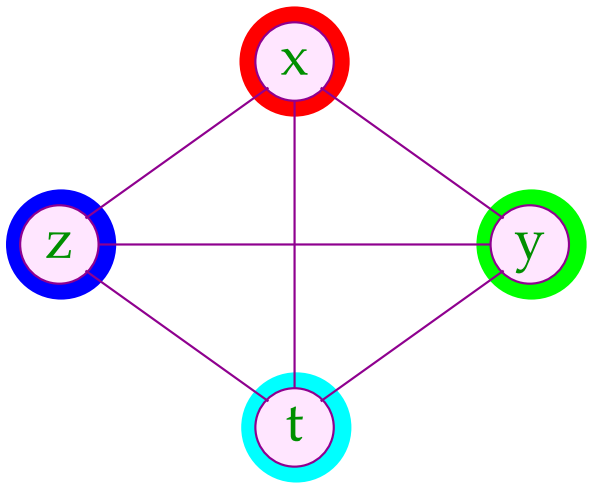
Beispiel:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



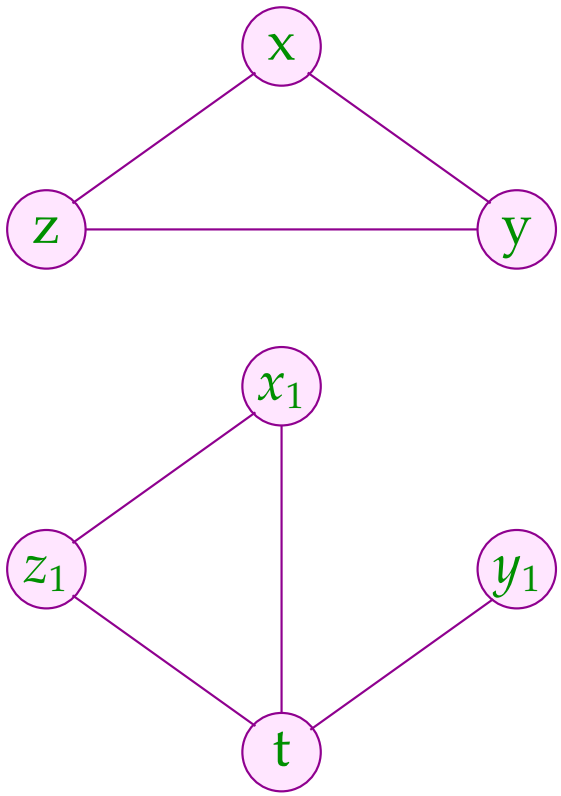
Beispiel:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



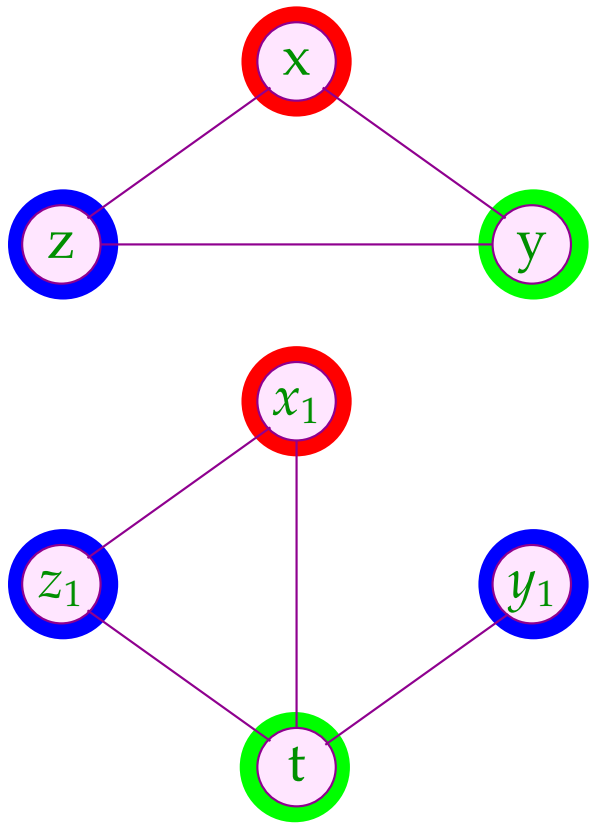
Die Lebendigkeitsbereiche von x und z können wir aufteilen:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	



Die Lebendigkeitsbereiche von x und z können wir aufteilen:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	



Technisch:

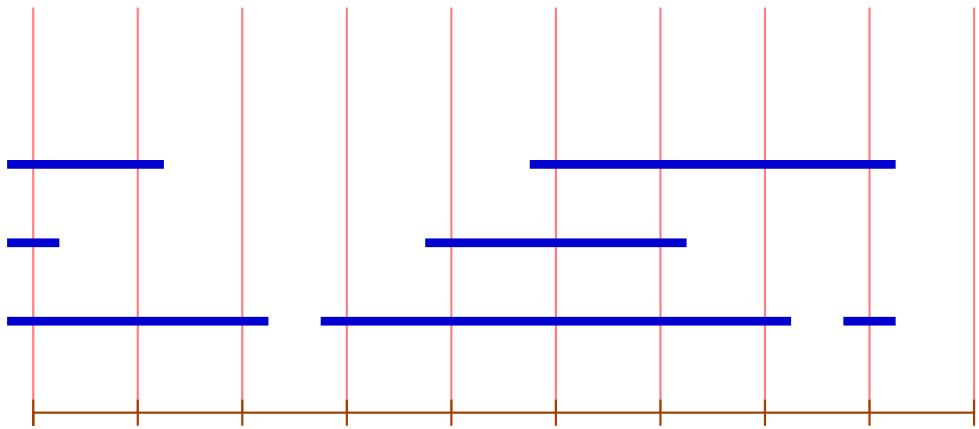
Eine Kante (u, lab, v) heißt x-transparent, falls lab keine Definition von x ist.

u, v gehören zum selben minimalen x-Lebendigkeitsbereich , falls $x \in \mathcal{L}[u] \cap \mathcal{L}[v]$ und u, v durch einen ungerichteten Pfad x-transparenter Kanten verbunden sind ...

Für jeden der minimalen x-Lebendigkeitsbereiche $L_1, \ldots, L_k$ für wir eine Variante von x ein :-)

Spezialfall: Basis-Blöcke

Die Interferenzgraphen für minimale Lebendigkeitsbereiche auf Folgen von Zuweisungen sind Intervall-Graphen:



Knoten $\Longleftrightarrow$ Intervall

Kante $\Longleftrightarrow$ gemeinsamer Punkt

Zu jedem Punkt können wir die Überdeckungszahl der inzidenten Intervalle angeben.

Satz:

maximale Überdeckungszahl

$\Longleftrightarrow$ Größe der maximalen Klique

$\Longleftrightarrow$ maximal nötige Anzahl Farben :-)

Graphen mit dieser Eigenschaft heißen perfekt ...

Eine minimale Färbung kann in polynomieller Zeit berechnet werden :-))

Idee:

- $\rightarrow$ Iteriere (konzeptuell) über die Punkte $0, \ldots, m-1$!
- $\rightarrow$ Verwalte eine Liste der aktuell freien Farben.
- $\rightarrow$ Beginnt ein neues Intervall, vergib die nächste freie Farbe.
- $\rightarrow$ Endet ein Intervall, gib seine Farbe frei.

Damit ergibt sich folgender Algorithmus:

```
free = [1,...,k];
for (i = 0; i < m; i++) {
    init[i] = []; exit[i] = [];
}
forall (I = [u,v] ∈ Intervals) {
    init[u] = (I :: init[u]); exit[i] = (I :: exit[v]);
}
    for (i = 0; i < m; i++) {
    forall (I ∈ exit[i]) free = color[I] :: free;
    forall (I ∈ init[i]) {
        color[I] = hd free; free = tl free;
    }
}
```

Diskussion:

- → Für Basis-Blöcke können wir eine optimale Aufteilung der Variablen auf eine Register ermitteln :-)
- → Das gleiche Problem ist bereits für einfache Schleifen (circular arc graphs) NP-schwierig :-(
- → Für beliebige Programme wird man deshalb eine Heuristik zum Graph-Färben einsetzen ...
- → Dieses Verfahren funktioniert besser, wenn wir die Lebendigkeitsbereiche maximal unterteilen :-)
- → Reicht die Anzahl der realen Register nicht aus, lagert man die überzähligen in einen festen Speicherbereich aus.
- → Man bemüht sich dabei, zumindest die in innersten Schleifen benutzten Variablen in Registern zu halten.

Interprozedurale Registerverteilung:

→ Für jede lokale Variable ist ein Eintrag im Kellerrahmen reserviert.

→ Vor dem Aufruf einer Funktion müssen die Register in den Kellerrahmen gerettet und danach restauriert werden.

→ Gelegentlich gibt es dafür Hardware-Unterstützung :-)

Dann ist ein Aufruf für alle Register transparent.

→ Verwalten wir Retten / Restaurieren selbst, können wir ...

- nur Register retten, deren Inhalte nach dem Aufruf noch benötigt werden :-)
- Register erst bei Bedarf restaurieren — und dann evt. in andere Register $\Longrightarrow$ Verkleinerung der Lebendigkeitsbereiche :-)

3.2 Instruktionen

Problem:

- unregelmäßige Instruktionssätze ...
- mehrere Adressierungsarten, die evt. mit arithmetischen Operationen kombiniert werden können;
- Register für unterschiedliche Verwendungen ...

Beispiel: Motorola MC68000

Dieser einfachste Prozessor der 680x0-Reihe besitzt

- 8 Daten- und 8 Adressregister;
- eine Vielzahl von Adressierungsarten ...

Notation	Beschreibung	Semantik
D_n	Datenregister direkt	D_n
A_n	Adressregister direkt	A_n
(A_n)	Adressregister indirekt	$M[A_n]$
$d(A_n)$	Adressregister indirekt mit Displacement	$M[A_n + d]$
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	$M[A_n + D_m + d]$
x	Absolut kurz	$M[x]$
x	Absolut lang	$M[x]$
$\#x$	Unmittelbar	x

- Der MC68000 ist eine 2-Adress-Maschine, d.h. ein Befehl darf maximal 2 Adressierungen enthalten. Die Instruktion:

  $$\texttt{add}\ \ D_1\ \ D_2$$

  addiert die Inhalte von D_1 und D_2 und speichert das Ergebnis nach und D_2 :-)

- Die meisten Befehle lassen sich auf Bytes, Wörter (2 Bytes) oder Doppelwörter (4 Bytes) anwenden.

  Das unterscheiden wir durch Anhängen von .B, .W, .D (Default: .W)

- Die Ausführungszeit eines Befehls ergibt sich (i.a.) aus den Kosten der Operation plus den Kosten für die Adressierung der Operanden ...

	Adressierungsart	Byte / Wort	Doppelwort
D_n	Datenregister direkt	0	0
A_n	Adressregister direkt	0	0
(A_n)	Adressregister indirekt	4	8
$d(A_n)$	Adressregister indirekt mit Displacement	8	12
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	10	14
x	Absolut kurz	8	12
x	Absolut lang	12	16
$\#x$	Unmittelbar	4	8

Beispiel:

Die Instruktion: move.B $8(A_1, D_1.W), D_5$

benötigt: $4 + 10 + 0 = 14$ Zyklen

Alternativ könnten wir erzeugen:

adda	$\#8, A_1$	Kosten:	$8+8+0=16$
adda	$D_1.W, A_1$	Kosten:	$8+0+0=8$
move.B	$(A_1), D_5$	Kosten:	$4+4+0=8$

mit Gesamtkosten 32 oder:

adda	$D_1.W, A_1$	Kosten:	$8+0+0=8$
move.B	$8(A_1), D_5$	Kosten:	$4+8+0=12$

mit Gesamtkosten 20 :-)

Achtung:

- Die verschieden Code-Sequenzen sind im Hinblick auf den Speicher und das Ergebnis äquivalent !
- Sie unterscheiden sich im Hinblick auf den Wert des Registers A_1 sowie die gesetzten Bedingungs-Codes !!
- Ein schlauer Instruktions-Selektor muss solche Randbedingungen berücksichtigen :-)

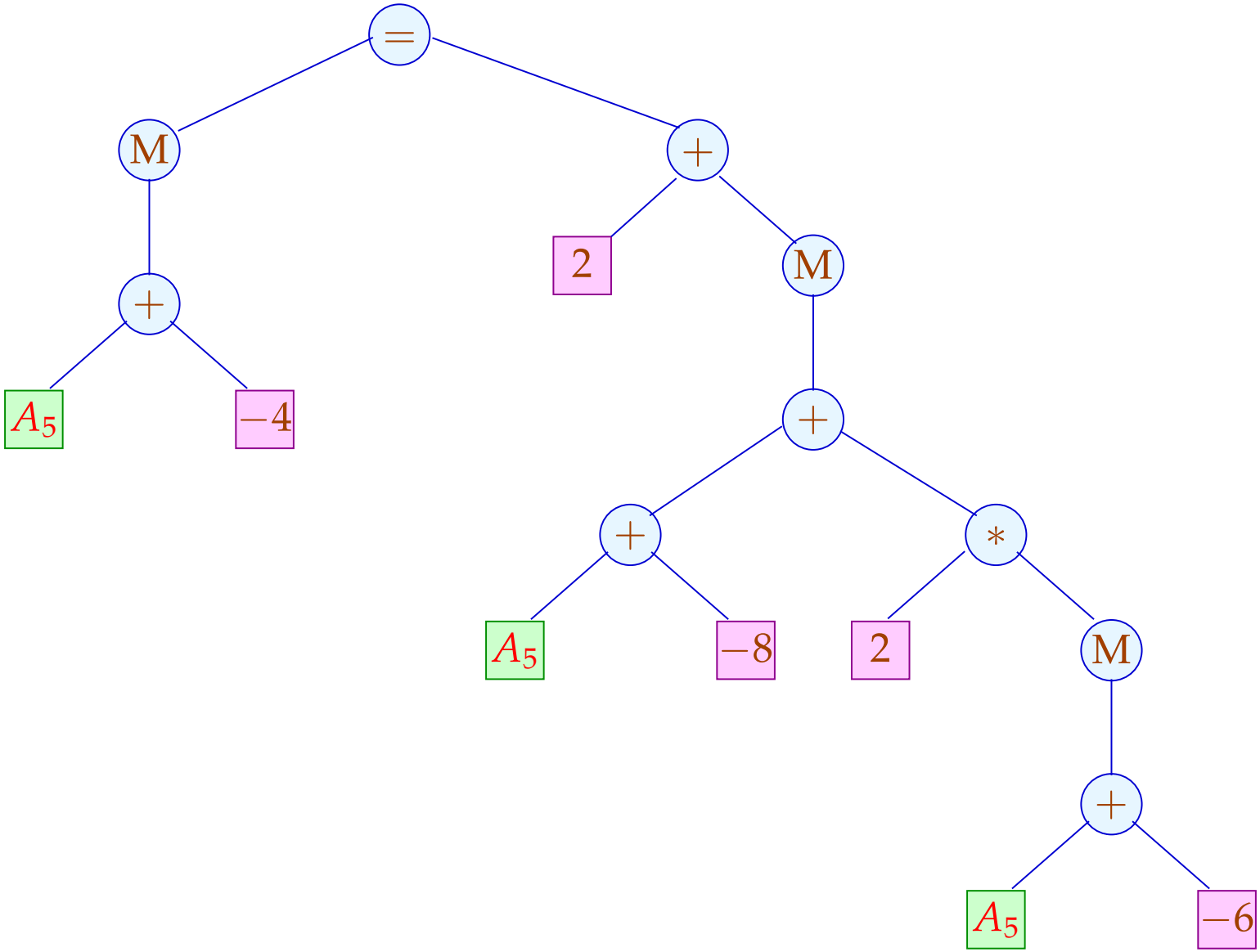
Etwas größeres Beispiel:

$$\text{int } b,\ i,\ a[100];$$
$$b = 2 + a[i];$$

Nehmen wir an, die Variablen werden relativ zu einem Framepointer A_5 mit den Adressen $-4, -6, -8$ adressiert.
Dann entspricht der Zuweisung das Stück Zwischen-Code:

$$M[A_5 - 4] = 2 + M[A_5 - 8 + 2 \cdot M[A_5 - 6]];$$

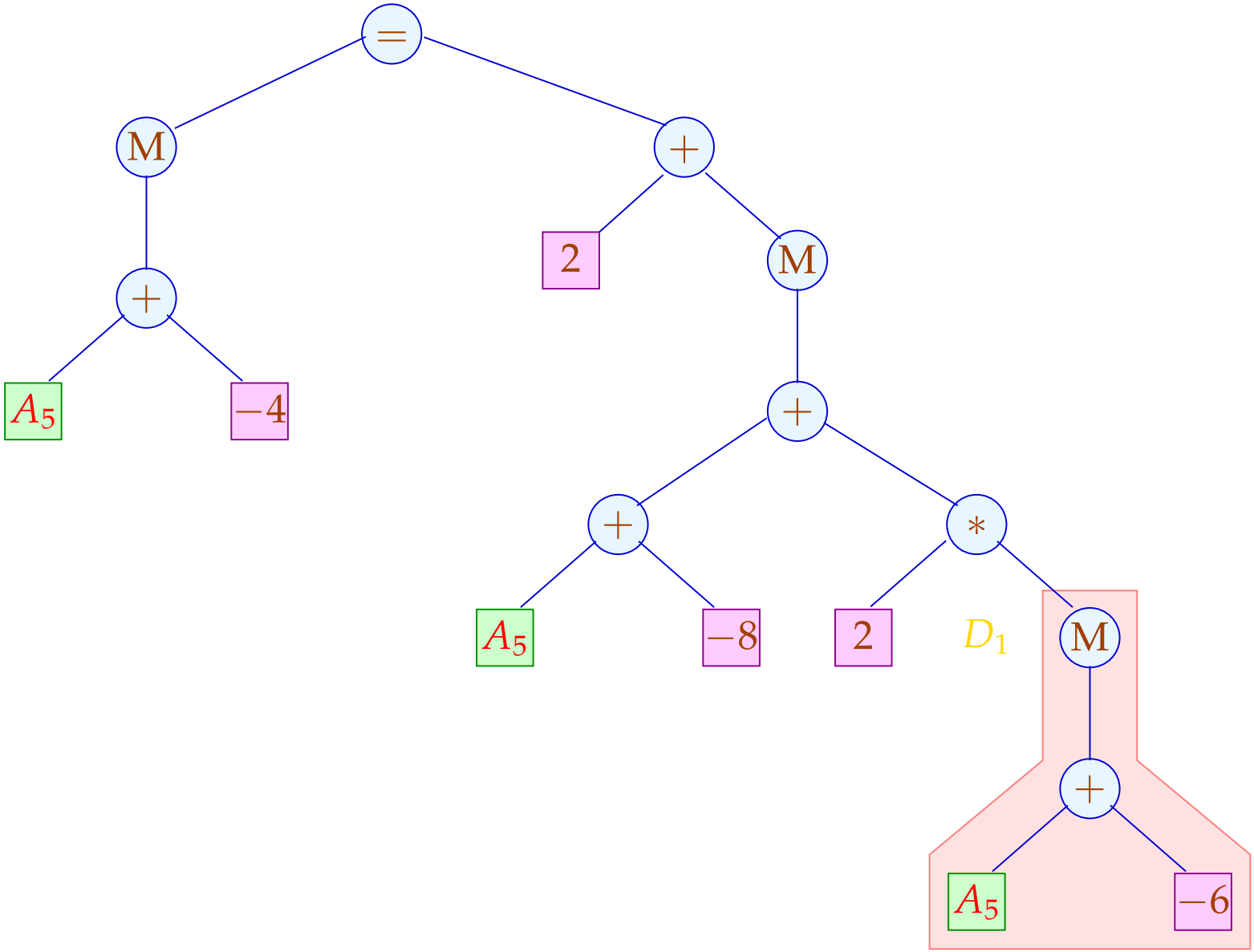
Das entspricht dem Syntaxbaum:

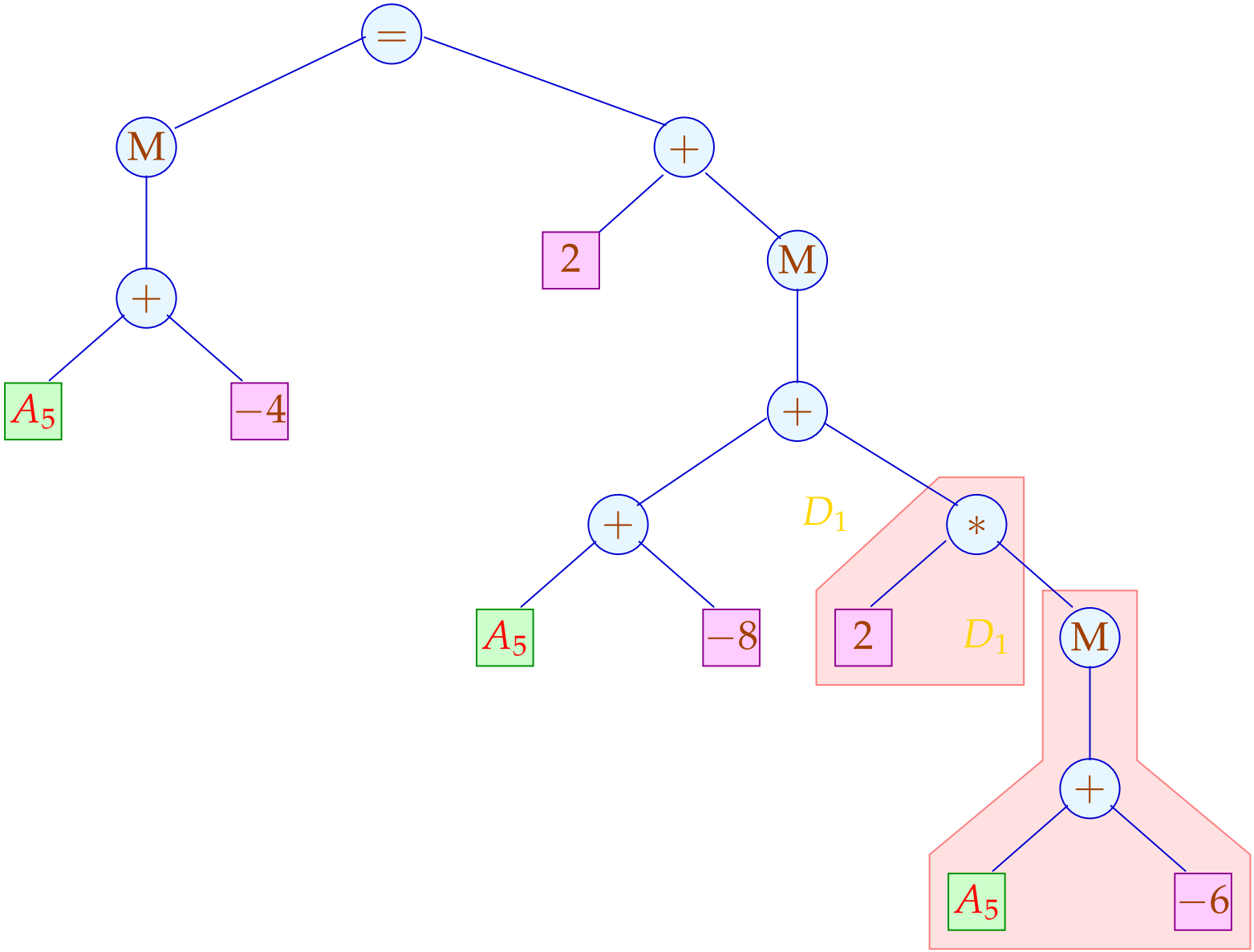

=
M
+
+
2
M
A_5
-4
+
*
+
A_5
-8
2
M
+
A_5
-6

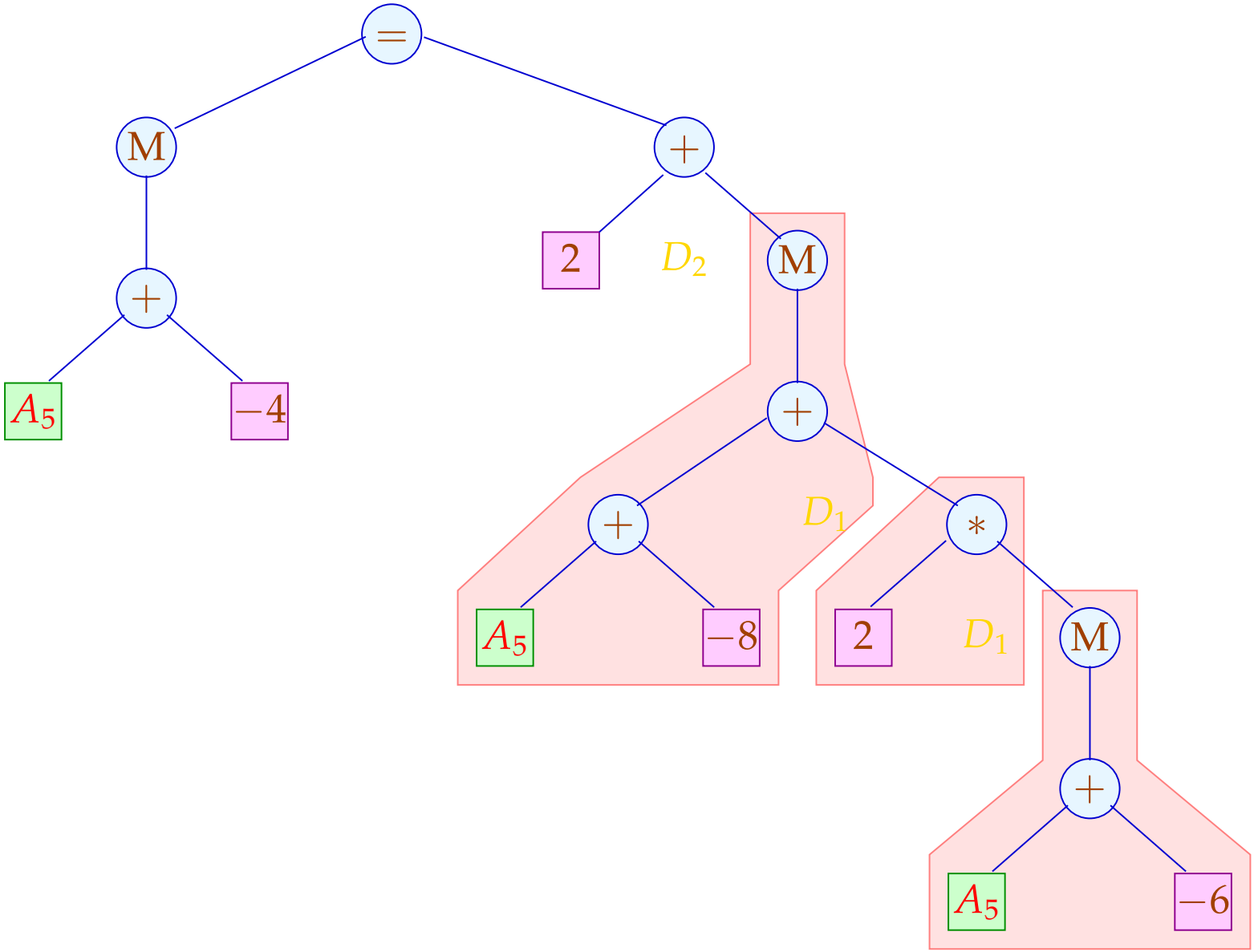
Eine mögliche Code-Sequenz:

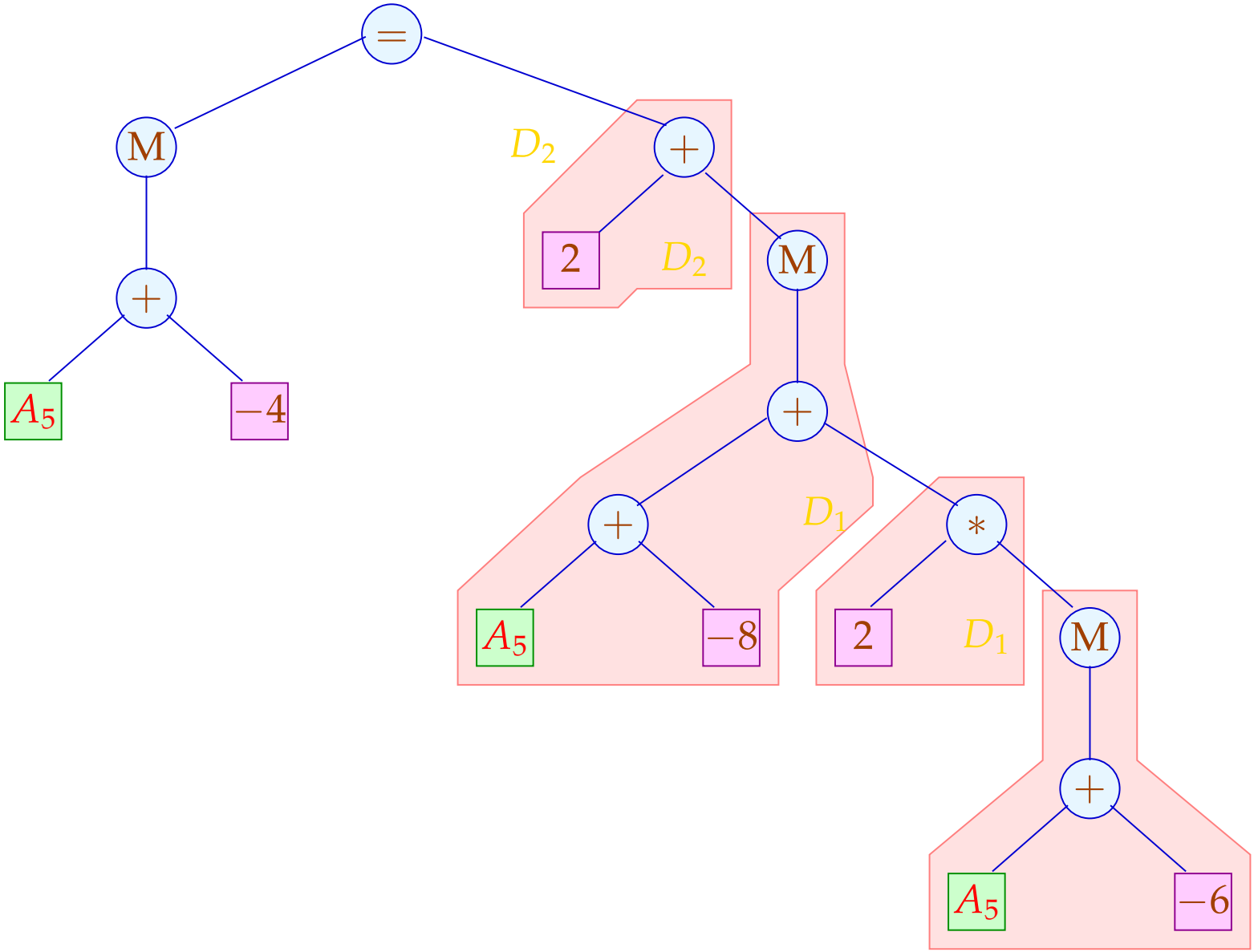
move	$-6(A_5),\ D_1$	Kosten:	12
add	$D_1,\ D_1$	Kosten:	4
move	$-8(A_5, D_1),\ D_2$	Kosten:	14
addq	$\#2,\ D_2$	Kosten:	4
move	$D_2,\ -4(A_5)$	Kosten:	12

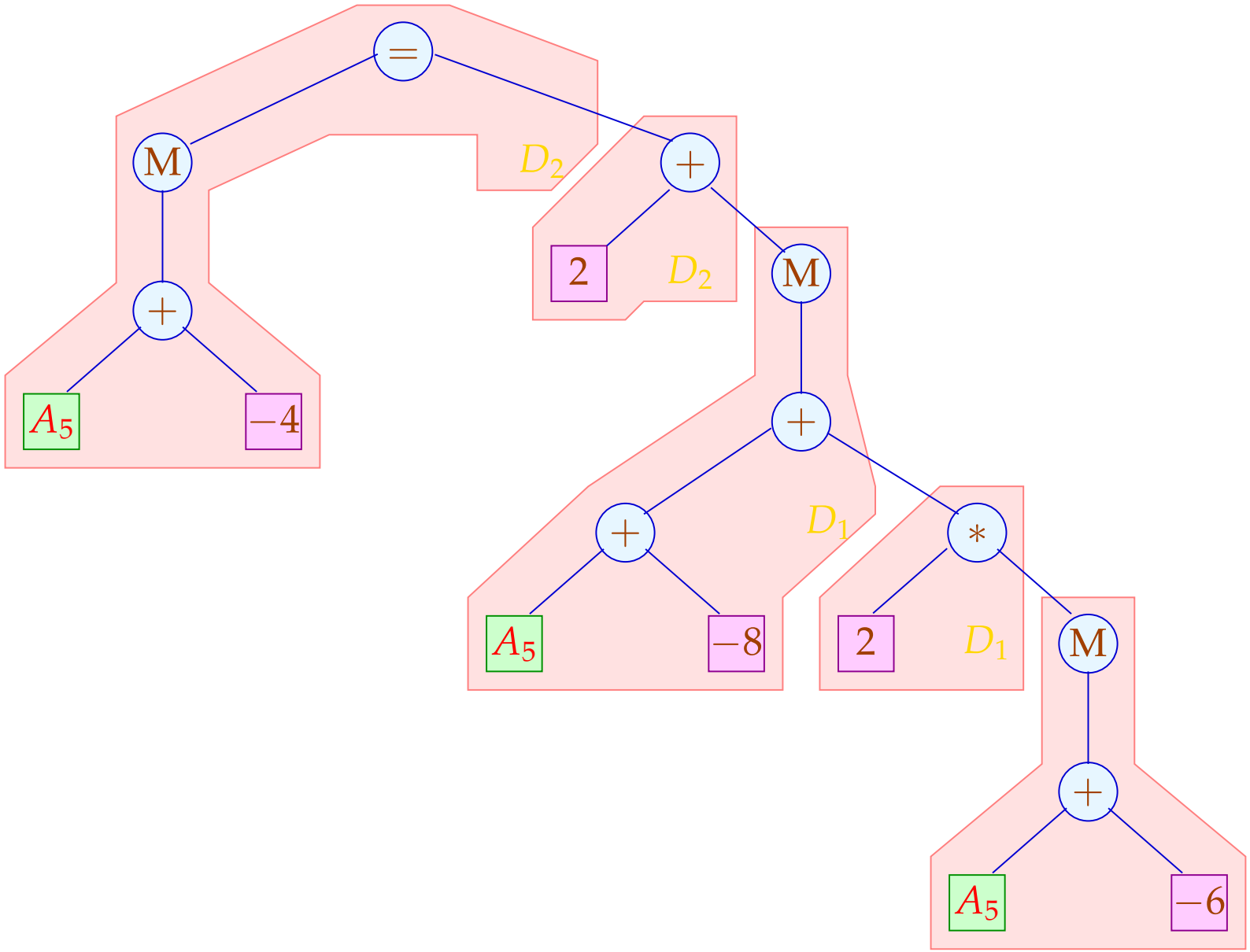
Gesamtkosten : 46


=
M
+
2
M
+
A_5
-4
+
+
*
A_5
-8
2
D_1
M
+
A_5
-6


=
M
+
2
M
+
A_5
-4
+
+
D_1
*
A_5
-8
2
D_1
M
+
A_5
-6


=
M
+
2
D_2
M
+
A_5
-4
+
D_1
+
*
A_5
-8
2
D_1
M
+
A_5
-6


=
M
+
A_5
-4
D_2
+
2
D_2
M
+
+
A_5
-8
D_1
*
2
D_1
M
+
A_5
-6


=
M
+
A_5
-4
D_2
+
2
D_2
M
+
+
A_5
-8
D_1
*
2
D_1
M
+
A_5
-6

Eine alternative Code-Sequenz:

move.L	$A_5,\ A_1$	Kosten:	4
adda.L	$\#-6,\ A_1$	Kosten:	12
move	$(A_1),\ D_1$	Kosten:	8
mulu	$\#2,\ D_1$	Kosten:	44
move.L	$A_5,\ A_2$	Kosten:	4
adda.L	$\#-8,\ A_2$	Kosten:	12
adda.L	$D_1,\ A_2$	Kosten:	8
move	$(A_2),\ D_2$	Kosten:	8
addq	$\#2,\ D_2$	Kosten:	4
move.L	$A_5,\ A_3$	Kosten:	4
adda.L	$\#-4,\ A_3$	Kosten:	12
move	$D_2,\ (A_3)$	Kosten:	8
Gesamtkosten :			124

Diskussion:

- → Die Folge ohne komplexe Adressierungsarten ist erheblich teurer :-(
- → Sie benötigt auch mehr Hilfsregister :-(
- → Die beiden Folgen sind nur äquivalent im Hinblick auf den Speicher — die Register haben anschließend verschiedene Inhalte ...
- → Eine korrekte Folge von Instruktionen kann als eine Pflasterung des Syntaxbaums aufgefasst werden !!!