5 Perspektiven

Herausforderungen:

- neue Hardware;
- neue Programmiersprachen;
- neue Anwendungen für Compiler-Technologie :-)

5.1 Hardware

Die Code-Erzeugung soll die Möglichkeiten der Hardware optimal ausnutzen ...

Herausforderungen:

Neue Hardware:

- → Speicher-Hierarchie mit unterschiedlich schnellen Caches für verschiedene Zwecke;
- → On-Board Nebenläufigkeit mit Pipelines, mehreren ALUs, spekulativer Parallelität, ...
- → Interaktion mit mächtigen Zusatzkomponenten wie Graphik-Karten ...

Eingeschränkte Hardware:

z.B. auf Chip-Karten, in Kühlschränken, Bremsanlagen, Steuerungen ...

$\Longrightarrow$ ubiquitous Computing

- $\rightarrow$ minimaler Energie-Verbrauch :-)
- $\rightarrow$ minimaler Platz :-)
- $\rightarrow$ Echtzeit-Anforderungen;
- $\rightarrow$ Korrektheit;
- $\rightarrow$ Fehler-Toleranz.

5.2 Programmiersprachen

Spezielle Features:

- mobiler Code;
- Nebenläufigkeit;
- graphische Benutzeroberflächen;
- Sicherheits-Komponenten;
- neue / bessere Typsysteme;
- Unterstützung für Unicode und XML.

Neue Programmiersprachen:

- XSLT;
- XQuery;
- Web-Services;
- anwendungs-spezifische Sprachen ...

5.3 Programmierumgebungen

Diverse Programmierhilfsmittel benutzen Compiler-Technologie ...

- syntax-gesteuerte Editoren;
- Programm-Visualisierung;
- automatische Programm-Dokumentation;
- partielle Codeerzeugung aus UML-Modellen;
- UML-Modell-Extraktion $\Longrightarrow$ reverse engineering
- Konsistenz-Überprüfungen, Fehlersuche;
- Portierung.

5.4 Neue Anforderungen

- Zuverlässigkeit
- Sicherheit

... im Rest der Vorlesung behandeln wir ausgewählte Themen, die bei der Code-Erzeugung für reale Maschinen relevant sind. Auch hier spielt die Idee der Generierung einzelner Komponenten eine wichtige Rolle:

6 Instruktions-Selektion

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 :-)

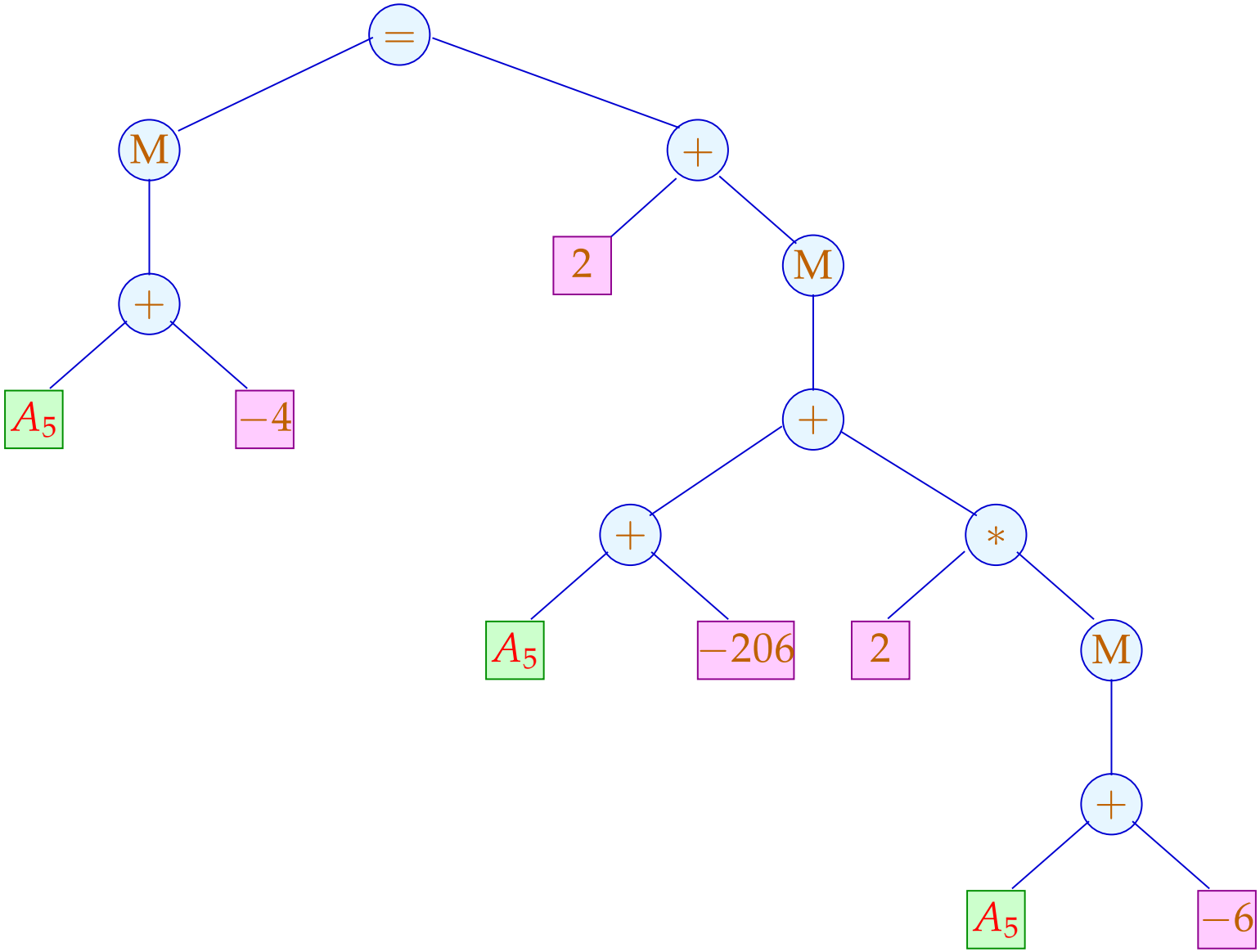
Etwas größeres Beispiel:

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

Nehmen wir an, die Variablen werden relativ zu einem Framepointer A_5 mit den Adressen $-4, -6, -206$ adressiert.
Dann entspricht der Zuweisung das Stück Zwischen-Code:

$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

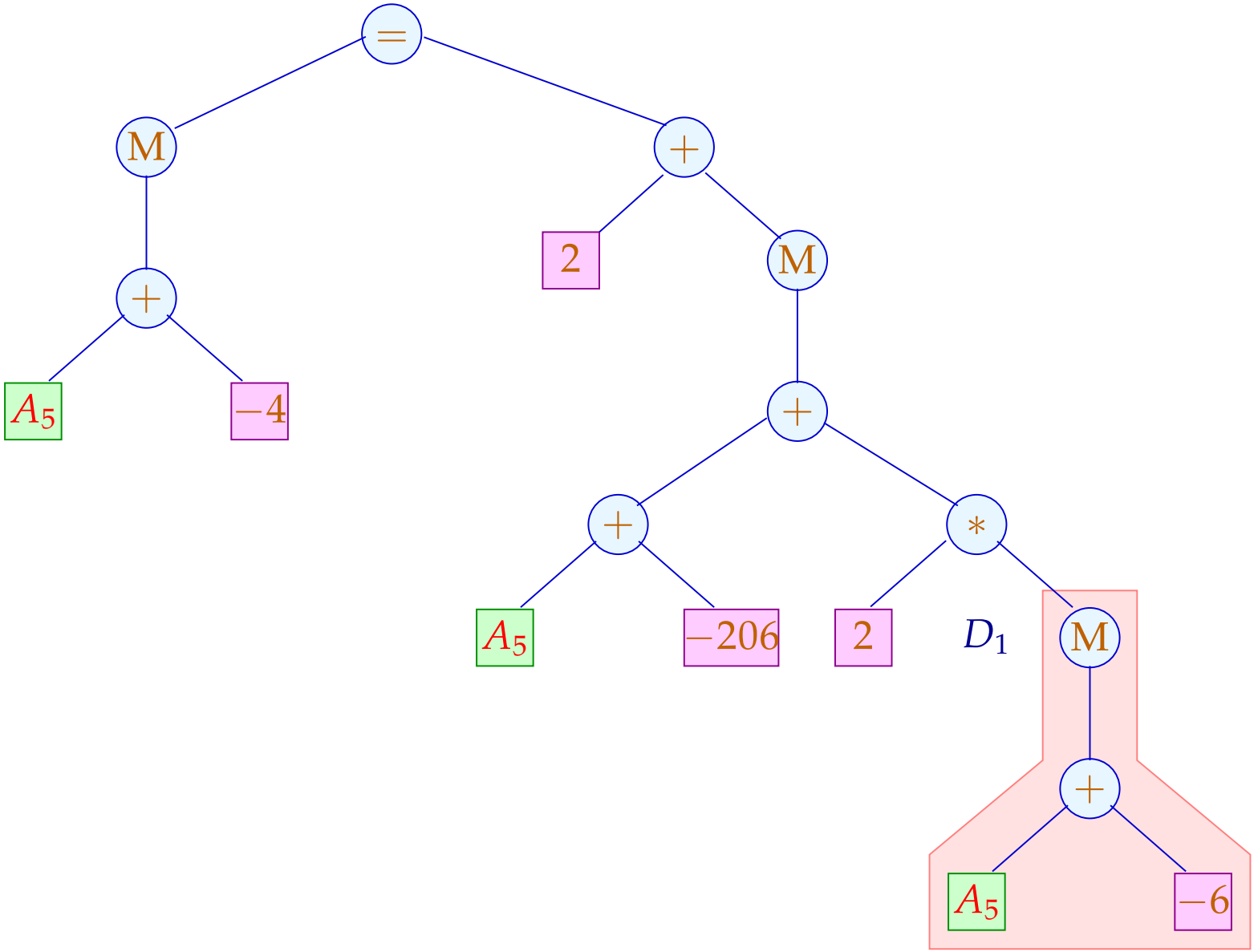
Das entspricht dem Syntaxbaum:

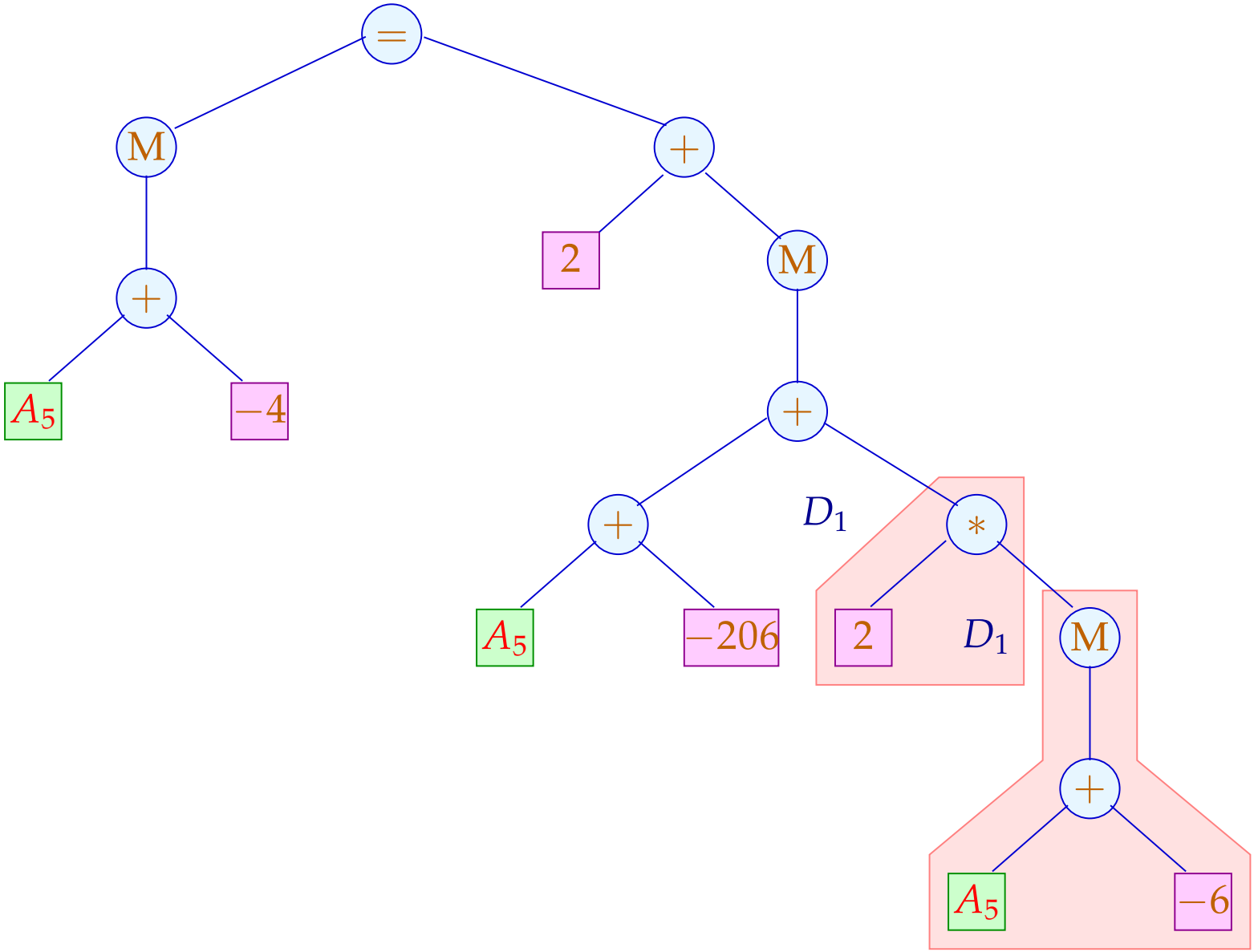

=
M
+
A_5
-4
2
M
+
+
*
A_5
-206
2
M
+
A_5
-6

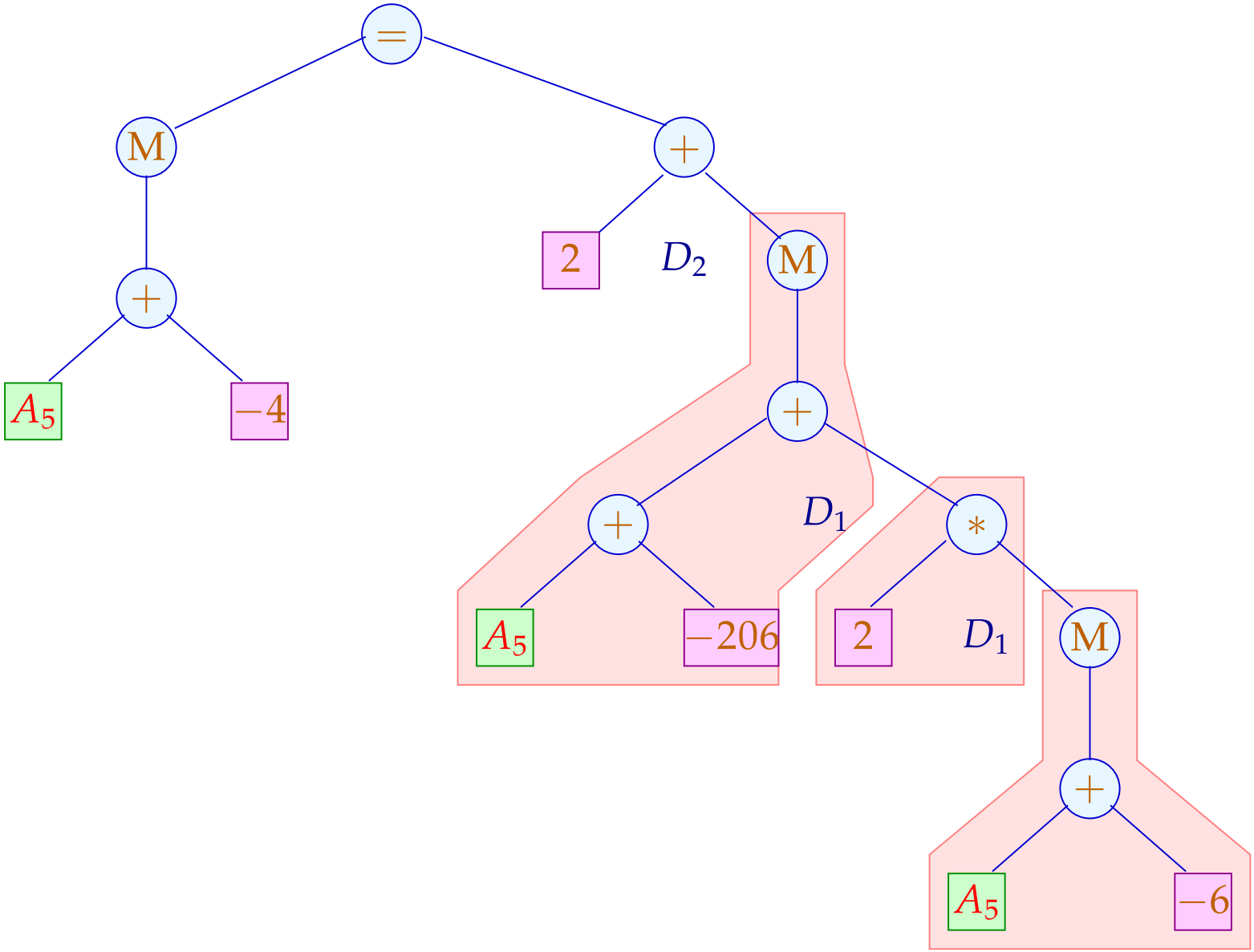
Eine mögliche Code-Sequenz:

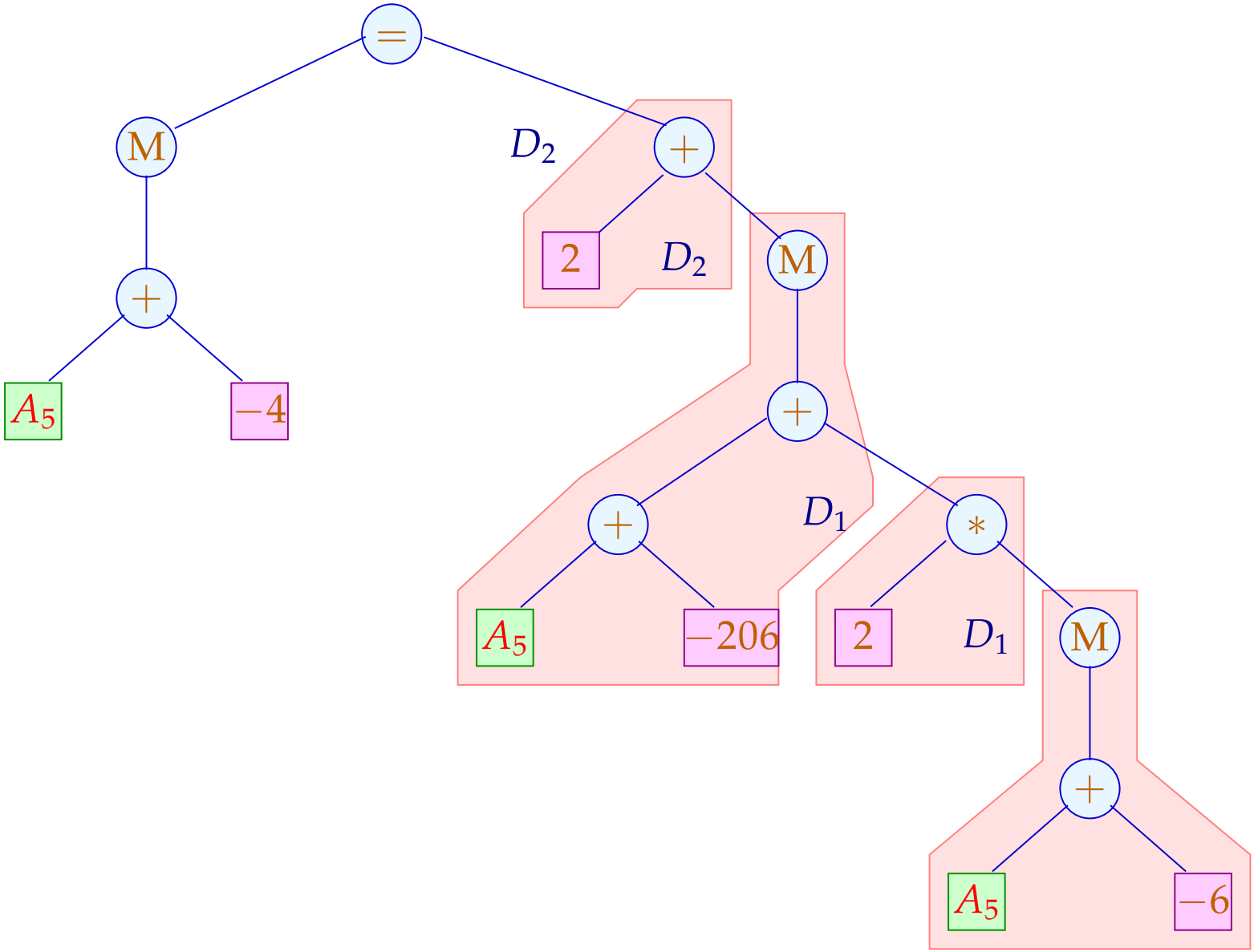
move	$-6(A_5),\ D_1$	Kosten: 12
add	$D_1,\ D_1$	Kosten: 4
move	$-206(A_5, D_1),\ D_2$	Kosten: 14
addq	$\#2,\ D_2$	Kosten: 4
move	$D_2,\ -4(A_5)$	Kosten: 12

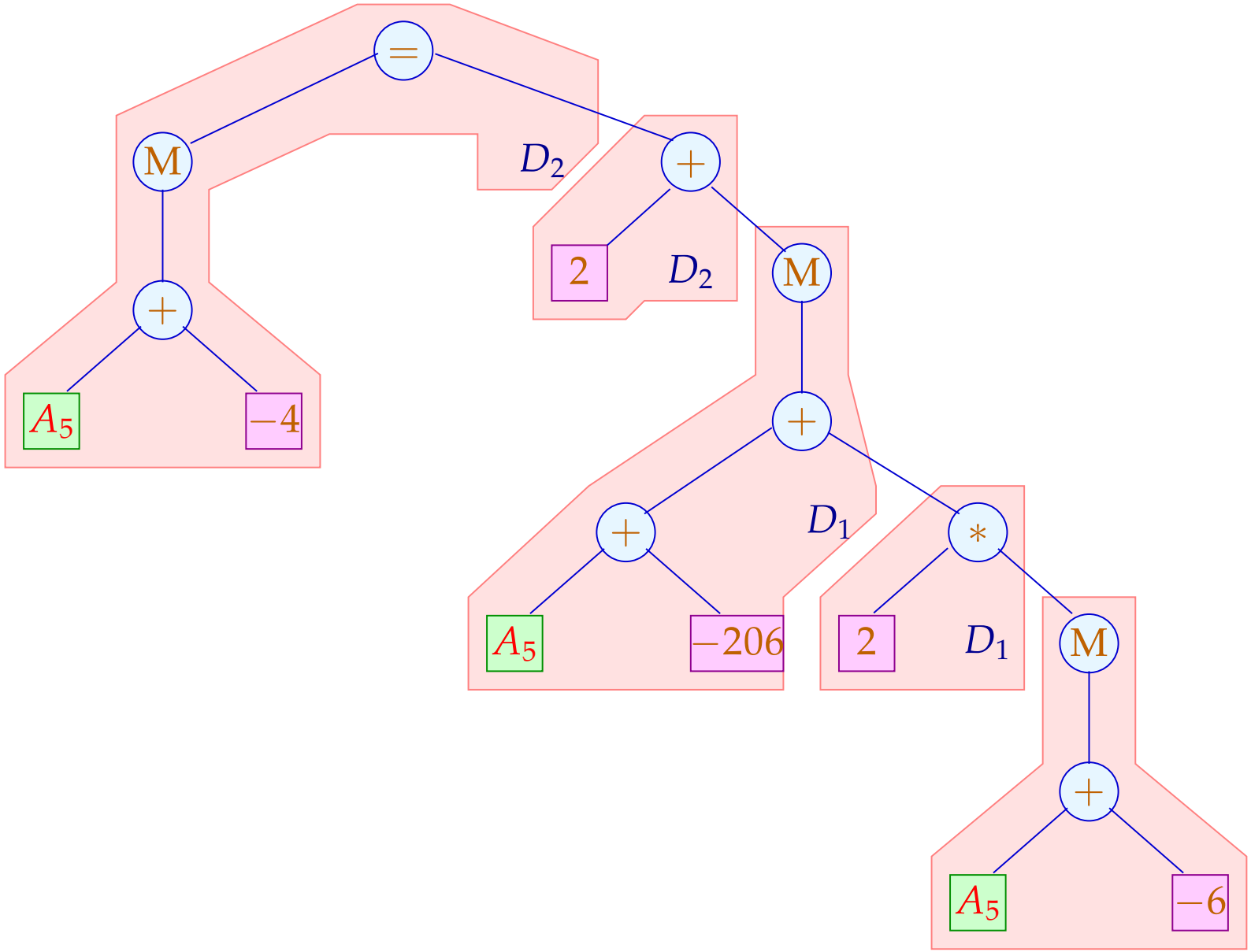
Gesamtkosten : 46


=
M
+
+
2
M
A_5
-4
+
*
+
A_5
-206
2
D_1
M
+
A_5
-6


=
M
+
2
M
+
A_5
-4
+
+
D_1
*
A_5
-206
2
D_1
M
+
A_5
-6


=
M
+
2
D_2
M
+
A_5
-4
+
D_1
+
*
A_5
-206
2
D_1
M
+
A_5
-6


=
M
+
A_5
-4
D_2
+
2
D_2
M
+
+
A_5
-206
D_1
*
2
D_1
M
+
A_5
-6


=
M
+
A_5
-4
D_2
+
2
D_2
M
+
+
A_5
-206
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	$\#-206,\ A_2$	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 !!!

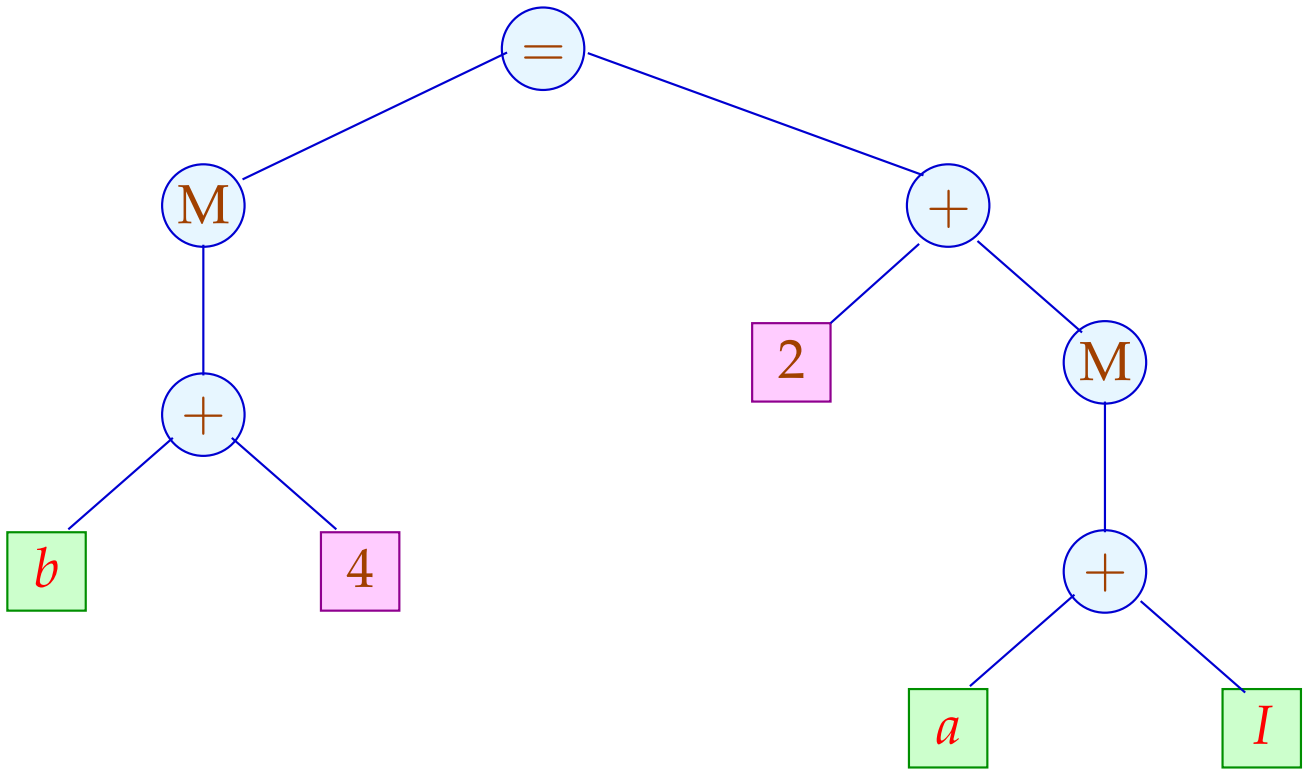
Genereller Ansatz:

- Wir betrachten Basis-Blöcke vor der Registerverteilung:

$$
\begin{aligned}
A &= a + I; \\
D_1 &= M[A]; \\
D_2 &= D_1 + 2; \\
B &= b + 4; \\
M[B] &= D_2
\end{aligned}
$$

- Wir fassen diese als Folge von Bäumen auf. Wurzeln:
  - Werte, die mehrmals verwendet werden;
  - Variablen, die am Ende des Blocks lebendig sind;
  - Stores.

... im Beispiel:



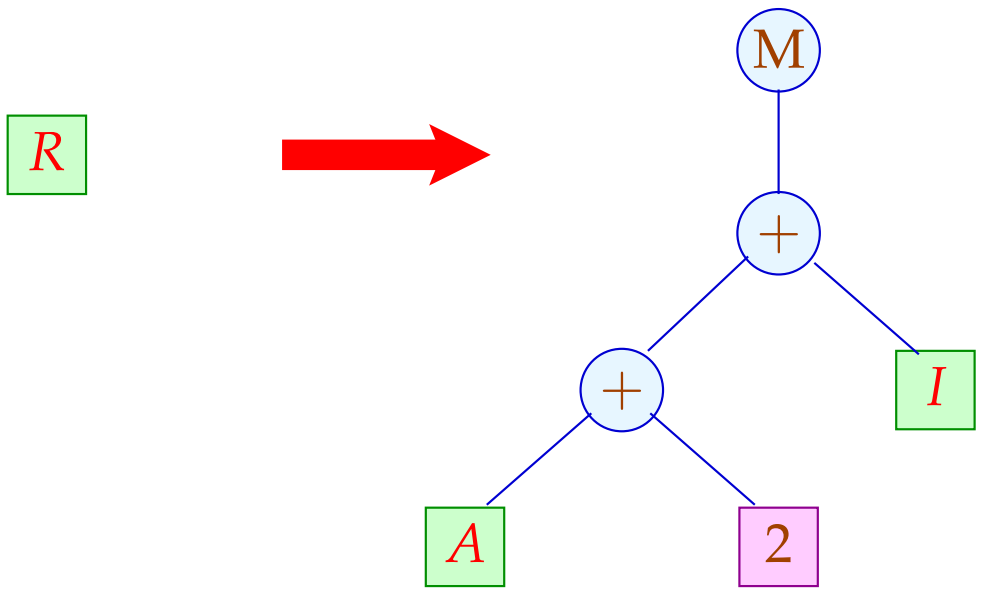
Die Hilfsvariablen A, B, D_1, D_2 sind vorerst verschwunden :-)

Idee:

Beschreibe den Effekt einer Instruktion als Ersetzungsregel auf Bäumen:

Die Instruktion: $R = M[A + 2 + D];$

entspricht zum Beispiel:

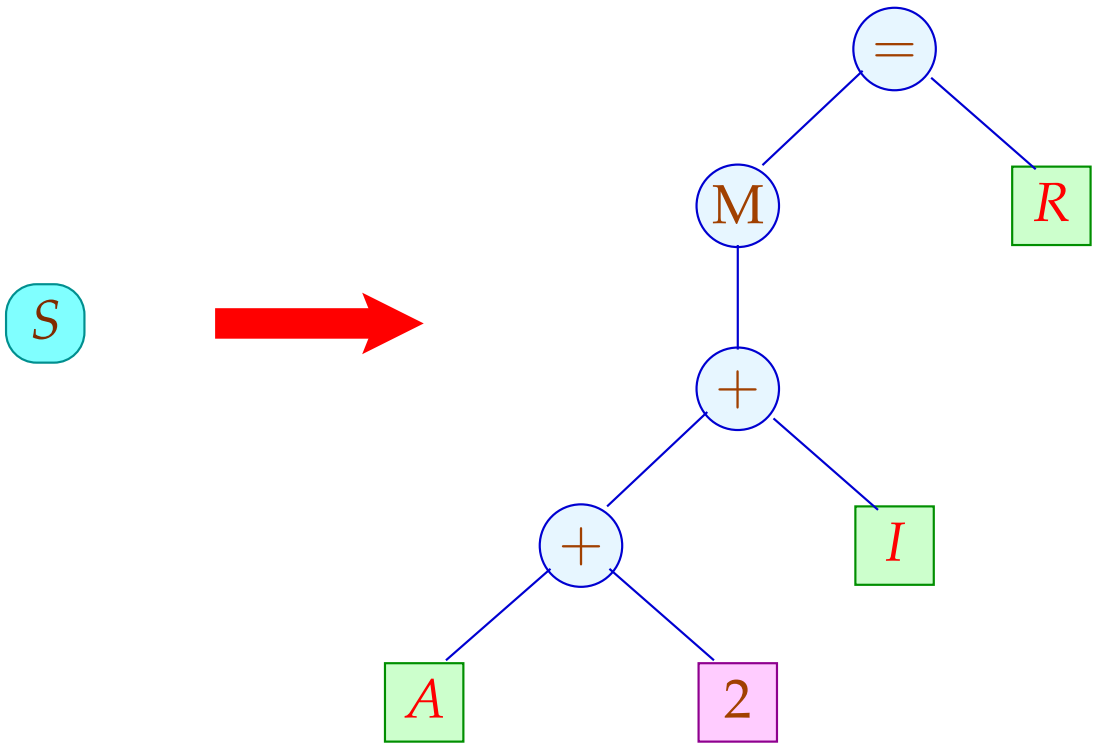


linke Seite	Ergebnisregister(klasse)
rechte Seite	berechneter Wert für Ergebnisregister
innere Knoten	• Load M • Arithmetik
Blätter	• Argumentregister(klassen) • Konstanten(klasse)

Die Grundidee erweitern wir (evt.) um eine Store-Operation.

Für die Instruktion: $M[A + 2 + D] = R;$

erlauben wir uns:



Die linke Seite S kommt nicht in rechten Seiten vor :-)

Spezifikation des Instruktionssatzes:

(1)	verfügbare Registerklassen	//	Nichtterminale
(2)	Operatoren und Konstantenklassen	//	Terminale
(3)	Instruktionen	//	Regeln

$\Longrightarrow$ reguläre Baumgrammatik