Codeerzeugung – Teil 1

Übersicht

- Abstrakte und reale Maschinen
  - Speicher in realen Maschinen
  - Codeerzeugung für reale Maschinen
- CISC und RISC
- Codeselektion
- Optimale Auswertungsordnung
- Codeerzeugung mit Registerzuordnung
- Registerzuteilung durch Graphfärbung

Lernziele

- Die wichtigsten Unterschiede zwischen abstrakten und realen Maschinen benennen können
- Die wichtigsten Eigenschaften von CISC- und RISC-Architekturen und die daraus resultierenden Fragestellungen auflisten können
- Die Probleme der Codeselektion kennen und erklären können, wie man sie lösen kann
- Die wichtigsten Prinzipien und Algorithmen der Registerzuteilung skizzieren können

Abstrakte und reale Maschinen

Thema

- Codeerzeugung für reale Maschinen
- Klassen realer Maschinen (mit unterschiedlichem Einfluss auf Codeerzeugung)
  - *CISC* ☛ (Complex Instruction Set Computer)
  - *RISC* ☛(Reduced Instruction Set Computer)

Abstrakte und reale Maschinen

- Bisher: Codeerzeugung für imperative Sprachen auf einer abstrakten Maschine
- Analog: Codeerzeugung für andere Sprachparadigmen (funktional, logisch, objektorientiert) wiederum auf (geeigneten) abstrakten Maschinen
- Nun: zusätzliche Probleme (und ihre Lösungen) beim Übergang auf reale Maschinen

Zunächst

- *Charakteristika* abstrakter Maschinen und Zusammenhang mit realen Maschinen in Bezug auf
  - Speicherorganisation
  - Befehlsvorrat
  - Ausdrucksauswertung

Sprachspezifische abstrakte Maschinen

Abstrakte Maschinen

- Auf spezielles Sprachparadigma (imperativ, funktional, logisch, ...) zugeschnitten
- Wechselseitige Verwendung schwierig (Sprachabhängigkeit!)
  - „Unpassende“, „Überflüssige“ oder „fehlende“ Maschinenbefehle
  - Unterschiedliche (Spezial-)Register

Charakteristika abstrakter Maschinen

- *Speicherorganisation*
  - Entsprechend der Lebensdauer verschiedener Klassen von Objekten
    - Keller
      - Für Objekte mit intervallartig geschachtelter Lebensdauer
      - Unterteilt in Rahmen mit statisch bestimmbarer Struktur
    - Halde: für übrige Objekte
  - Register für die Speicherorganisation (Kellerpegel, Haldenpegel, Rahmenzeiger)
- *Befehlsvorrat*
  - Komplexe Befehle zur einfachen (vollständigen oder teilweisen) Realisierung von Quellsprachenkonstrukten

Z.B. **mst**, **cupi**, **retp**

- *Ausdruckauswertung*
  - Mithilfe eines „kleinen“ Kellers, in dem stets Operanden und Ergebnisse vorliegen

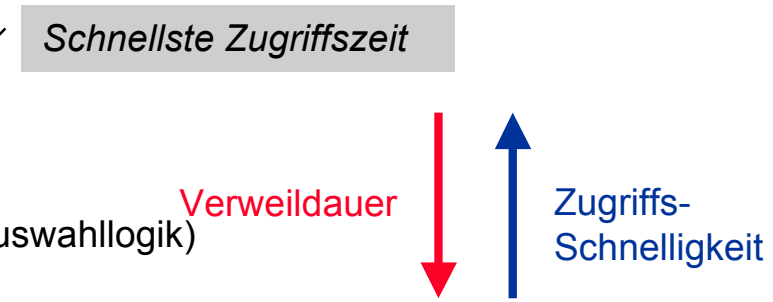
Universelle reale Maschinen

Reale Maschinen

- Müssen „universell“ sein; insbesondere erforderlich
  - Effiziente Realisierung der Speicherorganisation jeder abstrakten Maschine
  - Effiziente Simulation der jeweiligen Befehlsvorräte

Gemeinsame Charakteristika gängiger realer Rechner

- *Speicherorganisation*
  - Hierarchie verschiedener Speichertypen
    - *Prozessorregister* ☛(üblicherweise auf dem Chip)
    - Cache
    - *Hauptspeicher* ☛ (Zugriff über Systembus/Speicherauswahllogik)
    - Hintergrundspeicher (Zugriff über Betriebssystem)



  bestimmt durch Zugriffsschnelligkeit und Verweildauer der jeweiligen Inhalte
- *Befehlsvorrat*
  - Unterschiedliche Größe (30 - 1000), je nach konkreter Architektur (s.u.)
  - Verschiedene Befehlsarten (Berechnung, Transport, Sprünge, Kommunikation mit Peripherie,...)
- *Ausdrucksauswertung*
  - Meiste aktuell relevante Maschinen sind Registermaschinen (keine Kellermaschinen!)
  - Aufgabe der Codeerzeugung
    - Festlegung wo Zwischenergebnisse bei der Ausdrucksauswertung abgelegt werden
    - Geschickte Ausnutzung der Prozessorregister

Speicher in realen Maschinen

Hauptspeicher

- I.a. linearer (ansonsten unstrukturierter) Speicher zur Realisierung konzeptioneller Strukturen (Programmspeicher, Keller und Halde)
- Adressierbare Speichereinheiten verschieden groß (Bytes, „Worte“ = 2 Bytes, Doppelworte)

Prozessorregister

- I.a. *verschiedene Klassen von Registern* (bestimmt durch speicherbare Objekte und ausführbare Operationen), z.B.
  - Universalregister (Arithmetische + logische Operationen, Adressoperationen)
  - Gleitkommaregister (Operationen auf Gleitkommazahlen)
  - Getrennte Daten- und Adressregister
    - Auf Adressregistern: volle Adresslänge
    - Auf Datenregistern auch schnellere Operationen für Teilwerte
  - Basisregister (relative Adressierung von Speichersegmenten, z.B. Kellerrahmen)
  - Indexregister (für die Kombination mit anderen Registerinhalten zur Feldindizierung)
  - Register, die implizit in Operationen verwendet werden
    - Befehlszähler (program counter, PC)
    - Bedingungscode (condition code, CC) zur Aufnahme impliziter Vergleiche mit 0

Codeerzeugung für abstrakte/reale Maschinen

Ähnlichkeiten/Unterschiede in der Codeerzeugung

- *Kontrollstrukturen*
  - Übersetzung: i.w. identisch
  - Abstrakte Maschinen: Minimalsatz an bedingten und unbedingten Sprüngen
  - Reale Maschinen: Fülle bedingter Sprünge (spezielle Situationen, kurze/lange Distanzen)
- *Verwaltung von Rekursion*
  - Abstrakte Maschinen: Laufzeitkeller mit impliziter Verwaltung
  - Reale Maschinen: Kellerverwaltung durch reale Befehle realisiert
- *Adressierung von Variablen und Parametern*
  - Gemeinsam: Statisch vergebene Relativadressen, prinzipielle Zugriffsmechanismen
  - Abstrakte Maschinen: Objekte aller Typen in einem Maschinenwort
  - Reale Maschinen: Unterschiedliche Wortlängen, z.B.
    - Boolesche Werte und Zeichen: mehrere in ein Wort gepackt
    - Gleitkommawerte: i.a. mehrere Worte
    - Zeichenketten: in Halde abgelegt (dynamische Länge)
- *Auswertung von Ausdrücken*
  - Abstrakte Maschinen: Geeigneter Befehlssatz, um Übersetzungsprozess zu erleichtern
  - Reale Maschinen: Möglichst gute Ausnutzung der Leistungsfähigkeit der Prozessoren (bzgl. Registerzuteilung, Codeselektion, Instruktionsanordnung)

Codeerzeugung für reale Maschinen

Wichtigste Teilaufgaben der Codeerzeugung

- *Codeselektion* ☛
  - Auswahl geeigneter Maschinenbefehlsfolgen für „abstrakte Befehle“
  - Verwendung effizienterer Befehle für Spezialfälle
- *Registerzuteilung* ☛
  - Möglichst gute Ausnutzung der verfügbaren Register (Zugriffsschnelligkeit!) für
    - Speicherung von Variablenwerten und Zwischenergebnissen
    - Berechnung von Speicheradressen
    - Parameterübergabe
    - Rückgabe von Funktionswerten
  - Wechselseitige Abhängigkeit von Registerzuteilung und Codeselektion
- *Instruktionsanordnung und -umordnung* ☛
  - Ausschöpfung der Möglichkeiten zur Parallelverarbeitung durch geschickte Anordnung unabhängiger Maschinenbefehle

Optimale Registerzuteilung ist NP-vollständig

CISC (Complex Instruction Set Computer)

Charakteristische Merkmale

- Viele, teilweise komplexe Befehle
  - Viele und komplexe Adressierungsarten für effizienten Zugriff auf Datenstrukturen (z.B. Felder, Verbunde, Listen, Kellerrahmen)
  - Viele Varianten von Operationen für verschieden lange Operanden und Kombinationen von Operanden- und Resultatsorten
  - Ca. 1000 erlaubte Kombinationen von Operationscodes, Adressierungsarten, Größenangaben
  - Unterschiedliche Instruktionslängen (bis zu 9 Bytes)
  - Unterschiedliche Ausführungszeiten für Befehle (evtl. mehrere Takte pro Instruktion)
- Wenige Prozessorregister
- Reale CISC-Prozessoren

  Intel 80x86, Motorola 680x0, NSC 32xxx, ...

Wichtig bezüglich Codeerzeugung

- Gute Registereinteilung (schwierig!)
- Gute Codeselektion (basierend auf geeignetem Kostenmaß)

RISC (Reduced Instruction Set Computer) (1/3)

Charakteristische Merkmale

- Nur wenige, sehr einfache Befehle
  (Erhöhte Ausführungsgeschwindigkeit durch Vereinfachung der einzelnen Instruktionen)
  - Einheitliche Instruktionslängen
  - Einheitliche Ausführungszeit: I.w. eine Maschineninstruktion pro Befehlszyklus
  - Bei länger dauernden Instruktionen: *Fließbandverarbeitung* ☛(pipelining)
- Rechnende Operationen und Resultatablage nur auf Registern
- Wenige (1 bis 2) Adressierungsarten (Vereinfachung der Kontrolllogik)
- Kontrolle fest „verdrahtet“ (statt mikroprogrammiert)
- Viele Register (oft mehr als 100); gelegentlich mehrere Register gleichzeitig über „Registerfenster“ zugreifbar
- *Parallel arbeitende funktionale Einheiten* ☛ (und/oder Kommunikationslogik)
- Reale RISC-Prozessoren: SPARC, DEC Alpha, Power-PC, AMD 29k, ARC, ARM, Atmel AVR, Intel i860 and i960, MIPS, Motorola 88000,...

Wichtig bezüglich Codeerzeugung

- Registerzuteilung abhängig von Organisation der Register
- Codeselektion einfach

RISC (Reduced Instruction Set Computer) (2/3)

Befehlsfließband

- Pro Zeitpunkt: mehrere (aufeinander folgende) Befehle in versch. Ausführungsphasen, z.B.
  1. Befehl laden und decodieren (instruction fetch and decode)
  2. Operanden laden (operand fetch)
  3. Instruktion ausführen (instruction execution)
  4. Resultat in Zielregister schreiben

Neuere Rechner (z.B. Ultra SPARC III) haben bis zu 14 Stufen

- Verarbeitung
  - Für jede Phase im Fließband eine Stufe zuständig
  - Ausführungszeit pro Stufe: ein Zyklus
  - Aufteilung in Phasen ⇒ kleinere Zykluszeit

Ideal gefüttertes Fließband

		Zyklus						
		1	2	3	4	5	6	7
Fließ-	1	B_1	B_2	B_3	B_4			
band-	2		B_1	B_2	B_3	B_4		
stufe	3			B_1	B_2	B_3	B_4	
	4				B_1	B_2	B_3	B_4

- Beobachtungen
  - Alle Phasen sollten möglichst gleich lang sein
  - Ein Befehl pro Zyklus (nach „Anlaufzeit“ von 3 Zyklen)
  - Nur möglich, wenn keine „Kollisionen“, d.h.
    - Kein gleichzeitiger Zugriff aufeinander folgender Befehle auf dieselben Hardware-Ressourcen
    - Kein Befehl ein noch nicht bereitgestelltes Resultat eines vorangegangenen Befehls benötigt

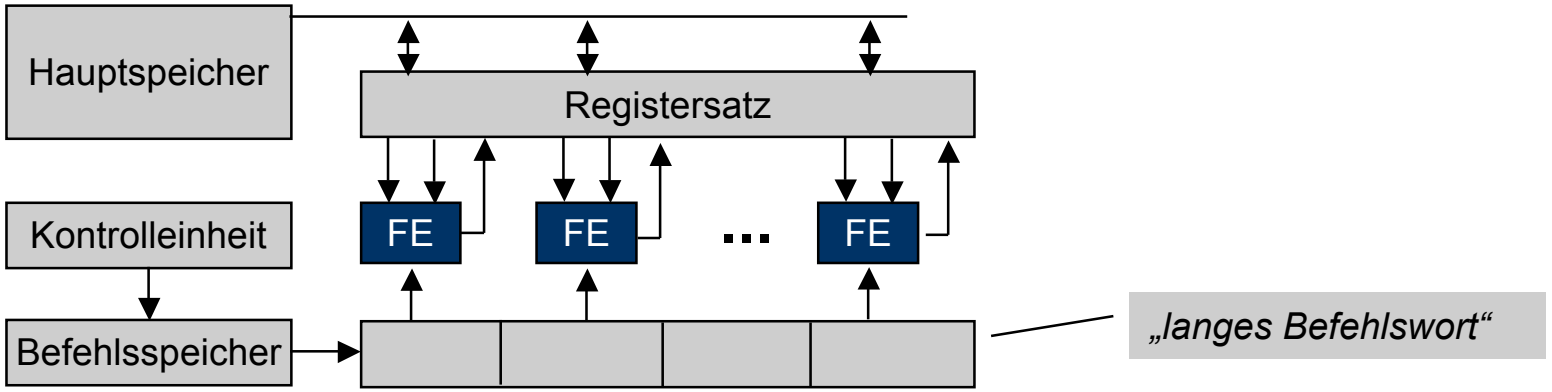
Aufgaben der Codeerzeugung

- (semantiktreue) Umordnung der Befehle, um o.g. Kollisionen zu vermeiden
- Evtl. Einfügen von NOP-Befehlen (no operation) zur Verzögerung der Befehlsauswertung

RISC (Reduced Instruction Set Computer) (3/3)

Parallel arbeitende funktionale Einheiten (FE)

- Mehrere parallel arbeitende FEs
- Registersatz (groß genug für gleichzeitiges Laden von Operanden durch alle FEs)
- Je zwei Eingabeverbindungen und eine Ausgabeverbindung zwischen FE und Registersatz
- „Lange Befehlswörter“ (mit je einem Befehl pro FE)
- Unabhängige Steuerung der FEs durch jeweilige Teilworte des Befehlsworts



Aufgabe des Übersetzers

- Packen der erzeugten Befehlsfolge in eine möglichst kurze Folge langer Befehlswörter, ohne Änderung der Semantik (schwierig; erfordert evtl. Datenabhängigkeitsanalyse)

Codeselektion (1/3)

Codeselektion

- Überführung einer Zwischendarstellung des zu übersetzenden Programms (Code für eine abstrakte Maschine) in möglichst effiziente Befehlsfolge für konkrete Zielmaschine
- Wichtig bei CISC-Maschinen (meist mehrere Möglichkeiten der Codeerzeugung)

Beispiel

- Motorola 68000: 8 Datenregister, 8 Adressregister (jeweils 4 Byte groß)
- Adressierungsarten ([A] stehe für „Inhalt von Adresse A“)
  - D_n Datenregister direkt: $[D_n]$
  - A_n Adressregister direkt: $[A_n]$
  - (A_n) Adressregister indirekt: $[[A_n]]$
  - $d(A_n)$ Adressregister indirekt mit Adressdistanzwert: $[[A_n]+d]$ (d ist 16-Bit-Konstante)
  - $d(A_n, I_x)$ Adressregister indirekt mit Index + Adressdistanzwert: $[[A_n]+[I_x]+d]$ (d ist 8-Bit-Konstante)
  - x absolut kurz: [x] (16-Bit-Konstante x ist Adresse einer Speicherzelle)
  - x absolut lang: [x] (32-Bit-Konstante x ist Adresse einer Speicherzelle)
  - #x unmittelbar: x (x ist Konstante)
- 2-Adress-Befehle, z.B.

  ADD D1, D2 (addiert Inhalte der Datenregister D1 und D2; speichert Resultat in D2)

Codeselektion (2/3)

Ausführungszeit eines Befehls

- Als Summe von
  - Ausführungszeiten für die Adressierung der Operanden und
  - Ausführungszeit für die Operation

Kurz: .B *bzw.* .W; *Ist auch Default*

Kurz: .L

	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 Adreßdistanzwert	8	12
$d(A_n, I_x)$	Adressregister indirekt mit Index und Adreßdistanzwert	10	14
x	absolut kurz	8	12
x	absolut lang	12	16
#x	unmittelbar	4	8

$= (A_n) + 4$

Beispiele

```
MOVE.B 8(A1, D1.W), D5
```

Kosten: 14 = 10 + 4

```
ADDA   #8, A1      12 = 4+8
ADDA   D1.W, A1     8 = 0+8
MOVE.B (A1), D5     8 = 4+4
```

Kosten: 28 = 8+20

```
ADDA   D1.W, A1     8 = 0+8
MOVE.B 8(A1), D5   12 = 8+4
```

Kosten: 20 = 8+12

- Beachte
  - Alternative Codesequenzen nur auf Speicher und Ergebnisregister D5 äquivalent;
  - Endzustand bezüglich Bedingungscode und Register A1 unterschiedlich
    ⇒ Codeselektor muss sicherstellen, dass Kontext jeweilige Codesequenz erlaubt

Codeselektion (3/3)

Beispiel

- Geg.: Anweisung b := a[i] + 2 wobei
  - b,i: Integer-Variable; a: Feld [0..] of integer
  - A5: Rahmenzeiger; Relativadressen 4, 6, 8, für b, i, a

Integer braucht 2 Worte

- Mögliche Codesequenz

i ⟶ D1	MOVE	6(A5), D1	Kosten: 12 = 8 + 4
2 * i ⟶ D1	ADD	D1, D1	Kosten: 4 = 0 + 4
Inhalt(a[i]) ⟶ D2	MOVE	8(A5, D1), D2	Kosten: 14 = 10 + 4
D2 + 2 ⟶ D2	ADDQ	#2, D2	Kosten: 4 = 4 + 0
D2 ⟶ b	MOVE	D2, 4(A5)	Kosten: 12 = 8 + 4

Gesamtkosten: 46

- Alternative Codesequenz

A5 ⟶ A1	MOVE	A5, A1	Kosten: 4
A1 + Adr(i) ⟶ A1	ADDA	#6, A1	Kosten: 12
Wert(i) ⟶ D1	MOVE	(A1), D1	Kosten: 8
2 * i ⟶ D1	MULU	#2, D1	Kosten: 44
A5 ⟶ A2	MOVE	A5, A2	Kosten: 4
A2 + Adr(a) ⟶ A2	ADDA	#8, A2	Kosten: 12
Adr(a+2i) ⟶ D2	ADDA	D1, A2	Kosten: 8
Wert(a[2i]) ⟶ D2	MOVE	(A2), D2	Kosten: 8
D2 + 2 ⟶ D2	ADDQ	#2, D2	Kosten: 4
A5 ⟶ A3	MOVE	A5, A3	Kosten: 4
A3 + Adr(b) ⟶ A3	ADDA	#4, A3	Kosten: 12
Zuweisung	MOVE	D2, (A3)	Kosten: 8

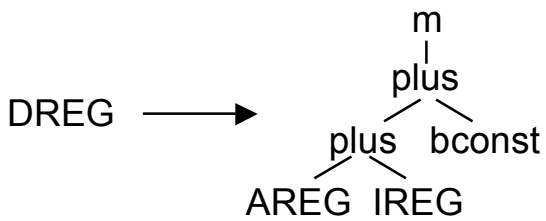
Gesamtkosten: 128


Generierung von Codeselektoren (1/2)

Beschreibung von Maschineninstruktionen

- Durch Regeln einer *regulären Baumgrammatik*
- Beispiel (Instruktion MOVE d(A_n, I_x), D_m)

DREG → m(plus(plus(AREG, IREG),bconst))



Maschinengrammatik

- Menge von Regeln der obigen Form, wobei
  - Terminalsymbole (klein geschrieben): (Teil-) Operationen, die die Instruktion ausführt
  - Nichtterminale (groß geschrieben): Lokationen bzw. Ressourcenklassen
  - Linke Seite: gibt an, wo Ergebnis der Instruktion abgelegt wird
  - Rechte Seite: „Bedeutung“ der Instruktion
- „Wortschatz“ einer Maschinengrammatik
  - Zwischendarstellungen (in Baumform) für Ausdrücke

Ableitungsbaum (bestehend aus Regelbezeichnungen)

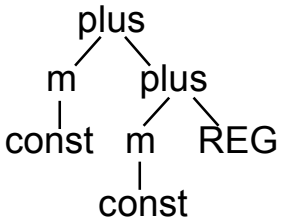
- Beschreibt eine Möglichkeit für zugehörigen Ausdruck Code zu erzeugen
- ⇒ Instruktionsauswahl umfasst:
  Analyseproblem für Zwischendarstellungsbäume bezüglich einer Maschinengrammatik

Generierung von Codeselektoren (2/2)

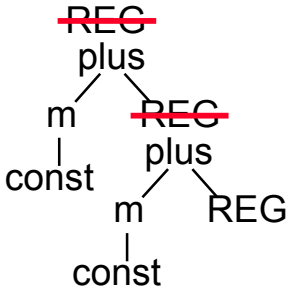
Beispiel

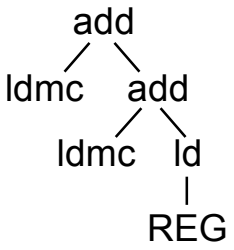
- Geg.: Regeln einer Baumgrammatik, Baum gemäß dieser Grammatik

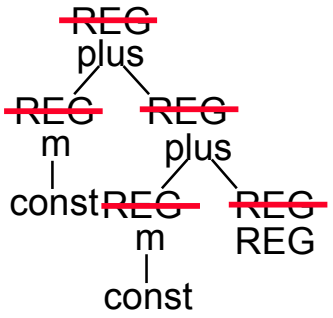
addmc:	REG	→	plus(m(const), REG)
addm:	REG	→	plus(m(REG), REG)
add:	REG	→	plus(REG, REG)
ldmc:	REG	→	m(const)
ldc:	REG	→	const
ld:	REG	→	REG



- Ableitungsbäume







Es gilt

- Maschinengrammatik i.a. mehrdeutig (d.h. verschiedene Instruktionsfolgen möglich)

Auswahl günstiger Instruktionsfolgen

- Annotation der Regeln mit Kosten (z.B. Anzahl der Maschinenzyklen)
- Auswahl des Ableitungsbaumes (= Instruktionsfolge) mit geringsten Kosten

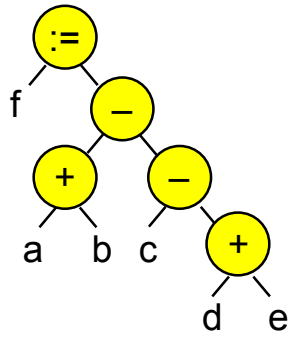
Optimale Auswertungsordnung

Problem

- Möglichst kurzer Code für Ausdrucksbäume
  (leicht auf Wertzuweisungen oder Basisblöcke erweiterbar)

Beispiel

- Baum für Wertzuweisung f := (a+b)-(c-(d+e))



Unterstelltes Maschinenmodell (RISC-Klasse)

- r allgemeine Register R_0, R_1, ..., R_{r-1}
- Direkte Korrespondenz zwischen Operationen im Ausdrucksbaum und Instruktionen
  (⇒ Codeselektion trivial)
- 2-Adress-Befehle mit folgendem Format (wobei V: Adresse einer Variable)
  - R_i := M[V] Laden
  - M[V] := R_i Speichern
  - R_i := R_i op M[V] Berechnen
  - R_i := R_i op R_j

Beispiel (für r = 2)

Befehlsfolge mit einer Zwischenspeicherung

R_0 := M[a]
R_0 := R_0 + M[b]
R_1 := M[d]
R_1 := R_1 + M[e]
M[t_1] := R_1
R_1 := M[c]
R_1 := R_1 – M[t_1]
R_0 := R_0 – R_1
M[f] := R_0

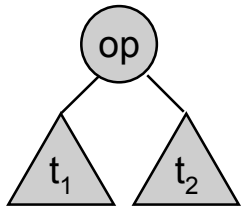
Befehlsfolge mit keiner Zwischen-speicherung

R_0 := M[c]
R_1 := M[d]
R_1 := R_1 + M[e]
R_0 := R_0 – R_1
R_1 := M[a]
R_1 := R_1 + M[b]
R_1 := R_1 – R_0
M[f] := R_1

Codeerzeugung mit Registerzuteilung (1/3)

Prinzip des allgemeinen Registerzuteilungsalgorithmus

- Geg.:
  - Baum t für Ausdruck e_1 op e_2 mit Teilbäumen t_1 für e_1 und t_2 für e_2
  - Benötigte Registeranzahl für t_1: r_1
  - Benötigte Registeranzahl für t_2: r_2



Insgesamt benötigte Register

- Falls gesamte Registeranzahl $r \geq r_1 > r_2$, dann
  - Nach Auswertung von t_1: r_1-1 Register frei (1 Register hält das Ergebnis)
  - Genügend Register für Auswertung von t_2
  - Gesamtzahl r_t benötigter Register: r_1
- Falls gesamte Registeranzahl $r \geq r_1 = r_2$, dann
  - Gesamtzahl r_t benötigter Register: $r_1 + 1$
- Falls gesamte Registeranzahl $r_t > r$, dann
  - Ablegen von Zwischenergebnissen in Speicherzellen

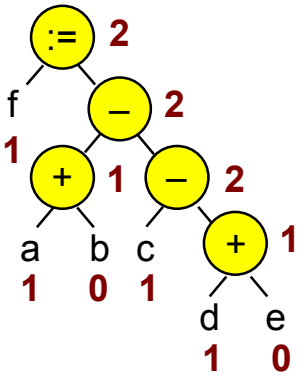
Codeerzeugung mit Registerzuteilung (2/3)

Algorithmus arbeitet in 2 Phasen

- *Markierungsphase* (Ermittlung des Speicherbedarfs der Teilbäume)
- *Generierungsphase* (Codeerzeugung durch berechneten Registerbedarf gesteuert)

Markierungsphase

- Markierung der Knoten des Baums mit ihrem *Registerbedarf* (Gesamtzahl der Register wird ignoriert) in einem bottom-up Durchlauf
  - „Linke Blätter“ (linke Knoten binärer Operationen): mit 1 markiert (da sie in ein Register geladen werden müssen)
  - „Rechte Blätter“: mit 0 markiert (da als Operanden benutzt)
  - Registerbedarf für innere Knoten op(t_1, t_2)
    - max(r_1, r_2), falls $r_1 \neq r_2$ (r_i = Registerbedarf für t_i)
    - $r_1 + 1$, falls $r_1 = r_2$



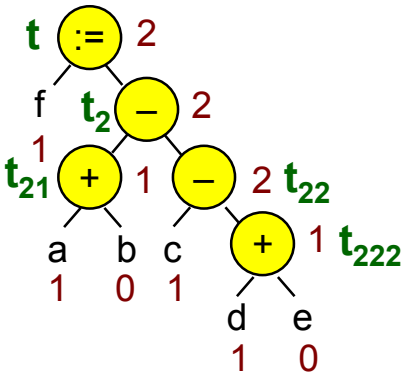
Generierungsphase

- Erzeugung *optimalen Codes* in einem (post-order) Baumdurchlauf
  - (1) Befehl **Op** (für Operation op an der Wurzel des Baums op(t_1, t_2)) wird nach dem Code für t_1 und t_2 erzeugt
  - (2) Reihenfolge der Bearbeitung der beiden Teilbäume entsprechend Registerbedarf
  - (3) Invariante für **Op**: Wert von t_1 liegt in einem Register vor (wird dann Operand in **Op**)
  - (4) Zwischenergebnisse für Teilbäume mit zu hohem Registerbedarf werden zwischengespeichert und wieder geladen

Codeerzeugung mit Registerzuteilung (3/3)

Algorithmus Gen_Opt_Code

```
var R: register; T: address;
var RST: stack of register; TST: stack of address;
proc GC (t: tree);
    case t of
    (leaf a, 1):                    (* linkes Blatt *)
        emit(top(RST)) := a);
    op((t1, r1), (leaf a, 0)):      (* rechtes Blatt *)
        GC(t1); emit(top(RST) := top(RST) Op a);
    op((t1, r1), (t2, r2)):
    cases
        r1 < min(r2, r):
        begin exchange(RST);
              GC(t2);  R := pop(RST);
              GC(t1);  emit(top(RST) := top(RST) Op R);
              push(RST, R); exchange(RST)  end;
        r1 ≥ r2 ∧ r2 < r:
        begin GC(t1);  R := pop(RST);
              GC(t2);  emit(R := R Op top(RST));
              push(RST, R)  end;
        r1 ≥ r2 ∧ r2 ≥ r:
        begin GC(t2);  T := pop(TST);  emit(M[T] := top(RST));
              GC(t1);  emit(top(RST) := top(RST) Op M[T]);
              push(TST, T)  end endcases endcase;
```



Ausdruck

R	=	R_1
T	=	
RST	=	R_0 R_1 R_2
TST	=	x_0 x_1 x_2

Keller für Register und Zwischenspeicher

Keller + Variable

Komplexitätsanalyse

- Algorithmus optimal bzgl. Anzahl erzeugter Befehle
  - innere Knoten: 1 Befehl
  - linke Blätter: 1 Befehl
  - Zwischenspeicherung, nur wenn nötig
- Allerdings: Mehrfachauswertung gemeinsamer Teilausdrücke

Erzeugter Code

```
R0 := c
R1 := d
R1 := R1 + e
R0 := R0 – R1
R1 := a
R1 := R1 + b
R1 := R1 – R0
R0 := f
R0 := R0 := R1
```

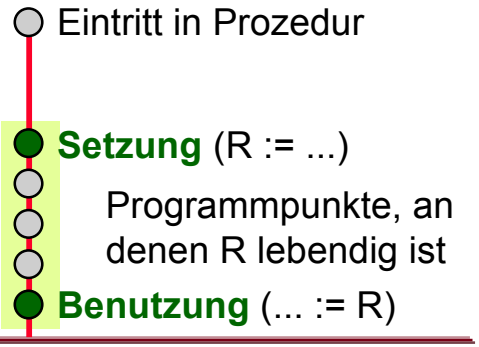
Registerzuteilung durch Graphfärbung (1/3)

Zielsetzung

- Bisher: Geschickte Registerverteilung für Ausdrucksbäume
- Nun: Globale Registervergabe für alle Berechnungen innerhalb einer Prozedur
- Eingabe
  - Zwischensprachenprogramm, in dem jeder Operation und jeder modifizierten Variablen ein *symbolisches Register* zugeteilt ist
- Aufgabe
  - Zuordnung der (im Prinzip unendlich vielen) symbolischen Register zu (beschränkt vielen) Registern der realen Maschine
- Problem
  - Ein reales Register kann nicht zwei verschiedenen symbolischen Registern zugeteilt werden, wenn beide gleichzeitig „lebendig“ sind (und nicht denselben Wert enthalten)

Lebendiges (symbolisches) Register R

- R **lebendig** an Programmpunkt p
  - p liegt auf Ausführungspfad zwischen Setzung und Benutzung
  - Zwischen p und Benutzung keine weitere Setzung
- **Lebensspanne** von R = {Punkte, an denen R lebendig ist}



Registerzuteilung durch Graphfärbung (2/3)

Kollisionsgraph

- Darstellung von Einschränkungen für die Zuteilung symbolischer Register an reale
- Zwei Lebensspannen symbolischer Register **kollidieren**, wenn eines der Register in der Lebenspanne des anderen gesetzt wird
- **Registerkollisiongraph**
  - Knoten: Lebenspannen von symbolischen Registern
  - (ungerichtete) Kanten: zwischen kollidierenden Lebensspannen

Damit

- Registerzuteilung ⇒ Graphfärbungsproblem
  - Kollisionsgraph mit k Farben färben (k = Anzahl der realen Register), wobei
  - Direkt durch Kante verbundene Knoten dürfen nicht dieselbe Farbe zugeteilt bekommen

Graphfärbungsproblem

- Für k > 2: NP-vollständig
- Bei der Registerzuteilung: heuristische Verfahren

Registerzuteilung durch Graphfärbung (3/3)

Idee des Graphfärbungsalgorithmus

- Suche in Graph G Knoten n mit Grad < k

  Für n bleibt eine Farbe übrig, unabhängig davon, wie die seiner Nachbarn sind

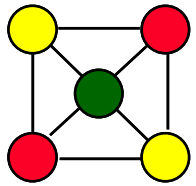
  - Entferne n und seine Kanten aus G
  - Wende Verfahren rekursiv auf reduzierten Graphen an, bis Graph leer (d.h. G ist k-färbbar)
  - Vergabe von Farben umgekehrt zur Reihenfolge des Knotenentfernens
- Falls kein Knoten mit Grad < k zu finden ist

  - Wähle (mit *Heuristik* ☛) einen Knoten aus, entferne ihn und seine Kanten und versuche so reduzierten Graphen mit k Farben zu färben
  - Entfernen des Knotens entspricht Zwischenspeichern des zugehörigen symbolischen Registers (für dieses Register: entsprechende Speicher-/ Ladeoperationen in das Programm einfügen)

Bemerkungen

- Knotengrad als Auswahlkriterium für Färbbarkeit nur hinreichend

  (Es gibt Graphen, die das Kriterium nicht erfüllen und doch k-färbbar sind)



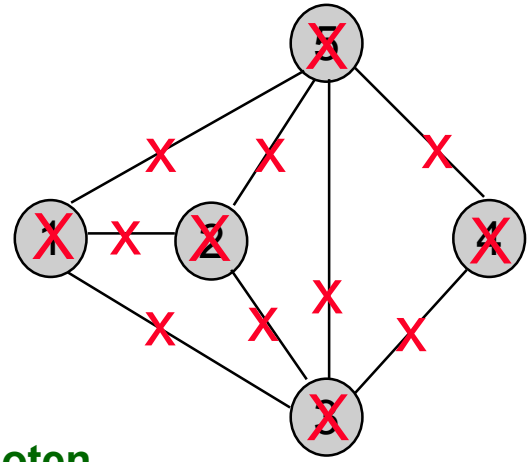
- *Auswahl-Heuristik* berücksichtigt
  - Grad der Knoten (hoher Grad steigert Chancen auf k-Färbbarkeit des reduzierten Graphen)
  - Kosten für die Zwischenspeicherung

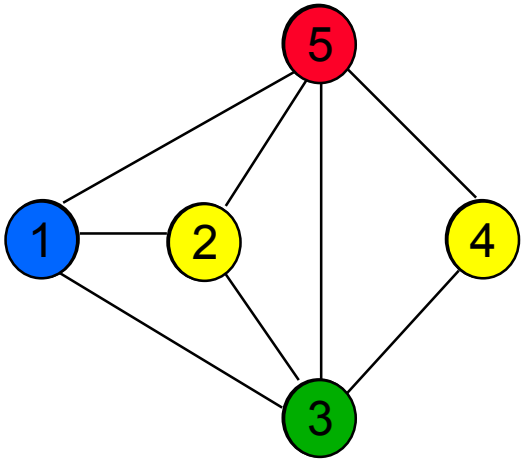
Mögliche Verbesserungen des Graphfärbungsalgorithmus

- Verschmelzen und Aufspalten von Lebensspannen (durch Änderungen im Programm)

Registerzuteilung durch Graphfärbung (3a/3)

Beispiel (k = 4)





Entfernte Knoten