Registervergabe: zwei Teilprobleme

1. Registerbelegung (register allocation):
   - welche Variablen werden in Registern gespeichert?
2. Registerzuweisung (register assignment):
   - in welchem Register wird eine Variable gespeichert?

- bisher wurden beide Teilprobleme integriert behandelt
  entweder
  (a) Annahme beliebig vieler Register (Verschiebung des Problems)
  oder
  (b) feste Anzahl von Registern und Spilling-Massnahmen

- bei verzweigtem Kontrollfluss: neue Herausforderung
  - Anzahl der Register kann von Registerzuweisung abhängen
  - Ziel: Minimierung dieser Anzahl

Lebendigkeit von Variablen

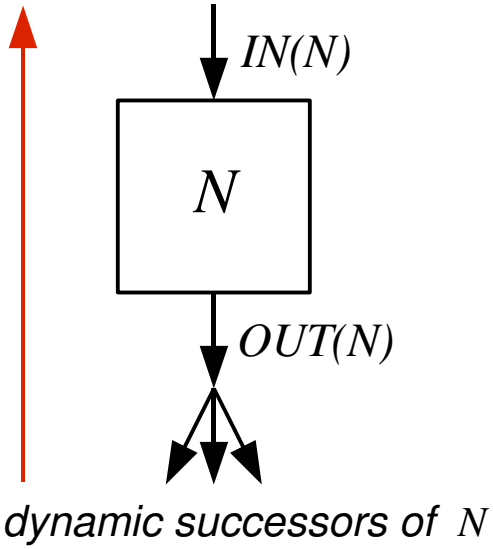
- eine Variable heisst *lebendig* zu den Zeiten, zu denen sie einen Wert enthält, der noch gebraucht wird (bzw. werden könnte):
  - nur gleichzeitig lebendige Variablen benötigen auch unterschiedliche Register
  - Lebendigkeitsanalyse reduziert die entstehenden Konflikte zwischen Variablen
- ein “Interferenzgraph” (auch “Konfliktgraph”) modelliert Konflikte:
  - jede Variable entspricht einem Knoten
  - zwei Knoten sind durch eine Kante verbunden, wenn sich die Lebendigkeitsintervalle der entsprechenden Variablen überlappen
  - Lösung der Registerzuweisung mittels Graphfärbung
- Einschub (Wdh. SIPS1): Liveness-Analyse

Wdh. aus SIPS1: Liveness-Analyse (1)

- Beispiel für Rückwärtsanalyse
- Definition (Liveness/Lebendigkeit):
  - eine Variable heisst *lebendig* an einem Knoten N im Kontrollflussgraphen gdw. sein Wert in N auf mindestens einem von N ausgehenden Pfad benutzt wird
  - eine Variable kann mehrere Lebenszeitintervalle haben
    - beginnend mit der Zuweisung eines Wertes
    - endend mit der letzten Benutzung dieses Wertes
- Motivation: (Aktionen am Ende eines Lebenszeitintervalls)
  - Speicherfreigabe für im Heap gespeicherte Datenobjekte
  - anderweitige Verwendung eines belegten Registers

Wdh. aus SIPS1: Liveness-Analyse (2)

- Rückwärtsanalyse entgegen der Richtung der Kanten des Datenflussgraphen
- Datenflussgleichungen
  - mergen von Eingangs- zu Ausgangsbedingungen
  - *KILL/GEN*: Ausgangsbedingungen → Eingangsbedingungen



$$OUT(N) = \bigcup_{M=\text{dynamic successor of } N} IN(M)$$

$$IN(N) = (OUT(N) \setminus KILL(N)) \cup GEN(N)$$

Figure 3.59 Backwards data-flow equations for a node N.

Wdh. aus SIPS1: Liveness-Analyse (3)

- Rückwärtsanalyse systematisch mit Datenflussgleichungen:

$$OUT(N) = \bigcup_{M=\text{dynamic successor of } N} IN(M)$$

$$IN(N) = (OUT(N) \setminus KILL(N)) \cup GEN(N)$$

Figure 3.59 Backwards data-flow equations for a node *N*.

- Wertzuweisungen an Variable V
  - KILL={ “V ist lebendig hier” }, GEN={ }
- Verwendung der Variablen V
  - KILL={ } , GEN={ “V ist lebendig hier” }
- beides (Vereinigung)
  - KILL={ “V ist lebendig hier” }, GEN={ “V ist lebendig hier” }

Wdh. aus SIPS1: Liveness-Analyse (4), Beispiel

- Bitmuster xy
  - `x=1` gdw. `x` ist lebendig
  - `y=1` gdw. `y` ist lebendig
- Merge ist bitweises logisches Oder
  - eine Variable ist lebendig, wenn ihr Wert auf einem der nachfolgenden Kontrollflusspfade verwendet wird

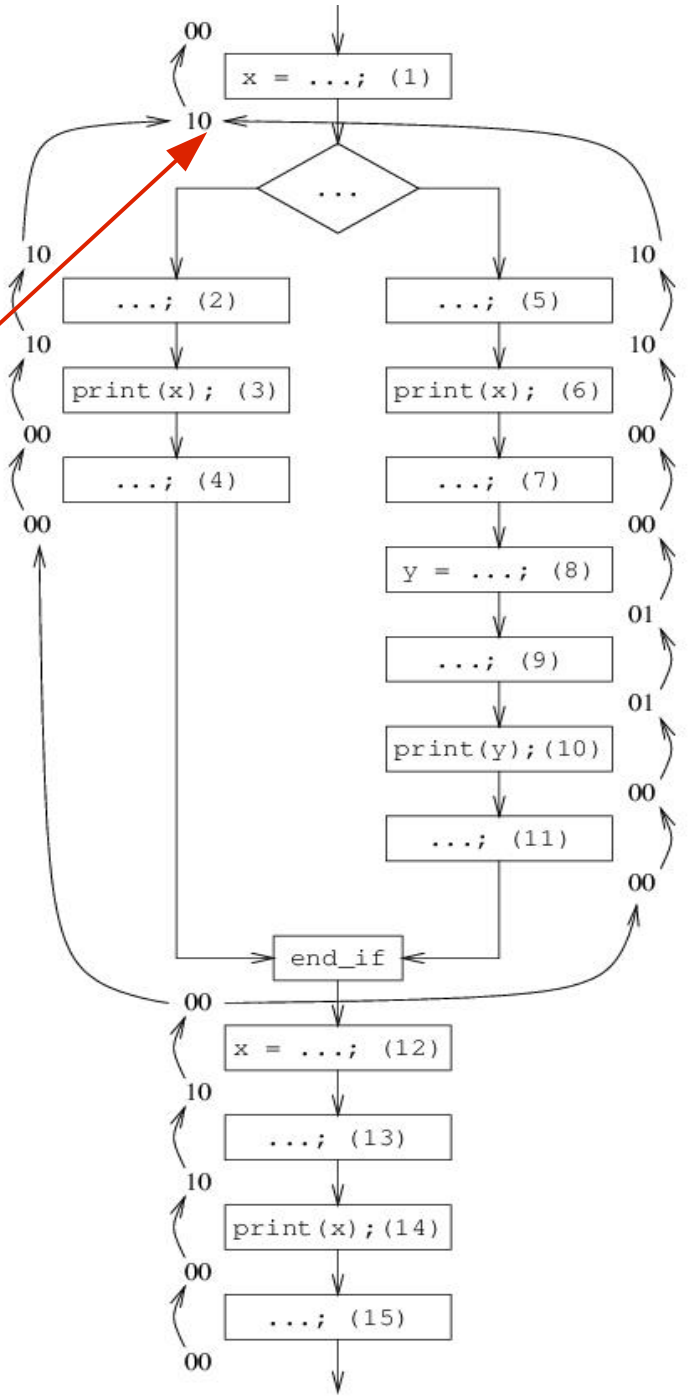


Figure 3.60 Live analysis for Figure 3.55 using backward data-flow equations.

Registerzuweisung mittels Graphfärbung (1)

- Vertex-Coloring-Problem
  - Instanz: ungerichteter Graph mit *n* Knoten
  - Frage: gibt es eine Färbung der Knoten mit höchstens *k* Farben, so dass nicht zwei Knoten gleicher Farbe durch eine Kante verbunden sind?
  - die Lösung muss zu jedem Knoten eine Farbe angeben
  - das Problem ist NP-vollständig
- zugehöriges Optimierungsproblem:
  - findet minimales *k, für das Lösung existiert*
  - liefert Lösung für dieses *k*
  - NP-hart (Reduktion des Entscheidungsproblems darauf)

Registerzuweisung mittels Graphfärbung (2)

Anwendung des Optimierungsproblems auf Registerzuweisung:

- Eingabe:
  - Knoten ≙ Variablen
  - zwischen zwei Knoten gibt es eine Kante gdw. sich die Lebenszeitintervalle der zugehörigen Variablen überlappen
- Ausgabe:
  - Farben ≙ Register
  - jeder Variablen (Knoten) wird ein Register (Farbe) zugewiesen
  - gleichzeitig lebendige Variablen werden *nicht* dem gleichen Register zugewiesen
  - die Anzahl benötigter Register ist minimal

Registerzuweisung, Beispielprogramm

```
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    print(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    print(e);
} else {
    f := 12;
    d := f + a;
    print(f);
}
print(d);
```

Figure 4.79 A program segment for live analysis.

`read()`
nicht-optimierbarer Ausdruck

`print(x)`
nicht-optimierbare Verwendung von `x`

Registerzuweisung, Lebenszeitintervalle

```
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    print(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    print(e);
} else {
    f := 12;
    d := f + a;
    print(f);
}
print(d);
```

Figure 4.79 A program segment for live analysis.

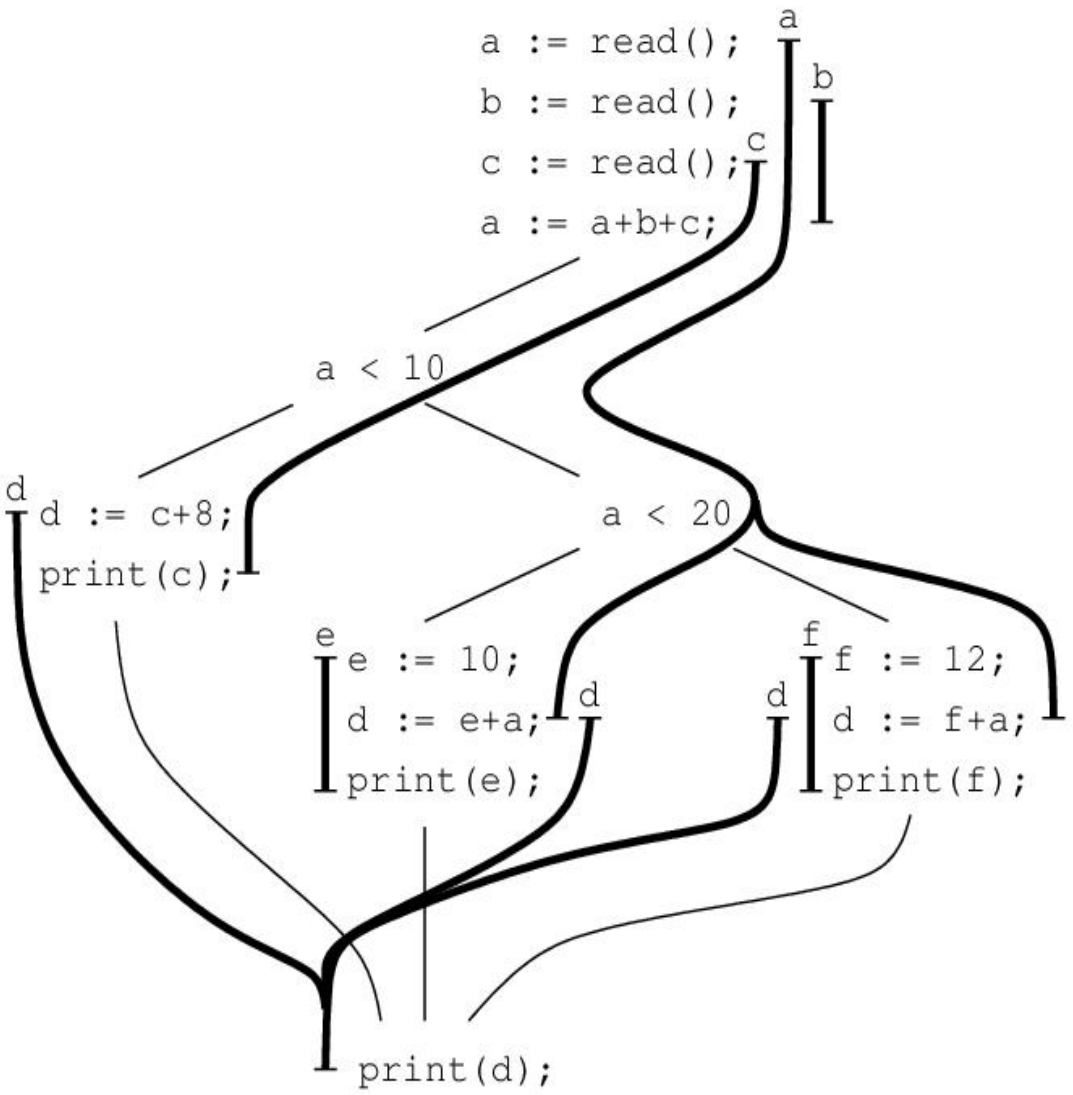


Figure 4.80 Live ranges of the variables from Figure 4.79.

Register-Interferenz-Graph

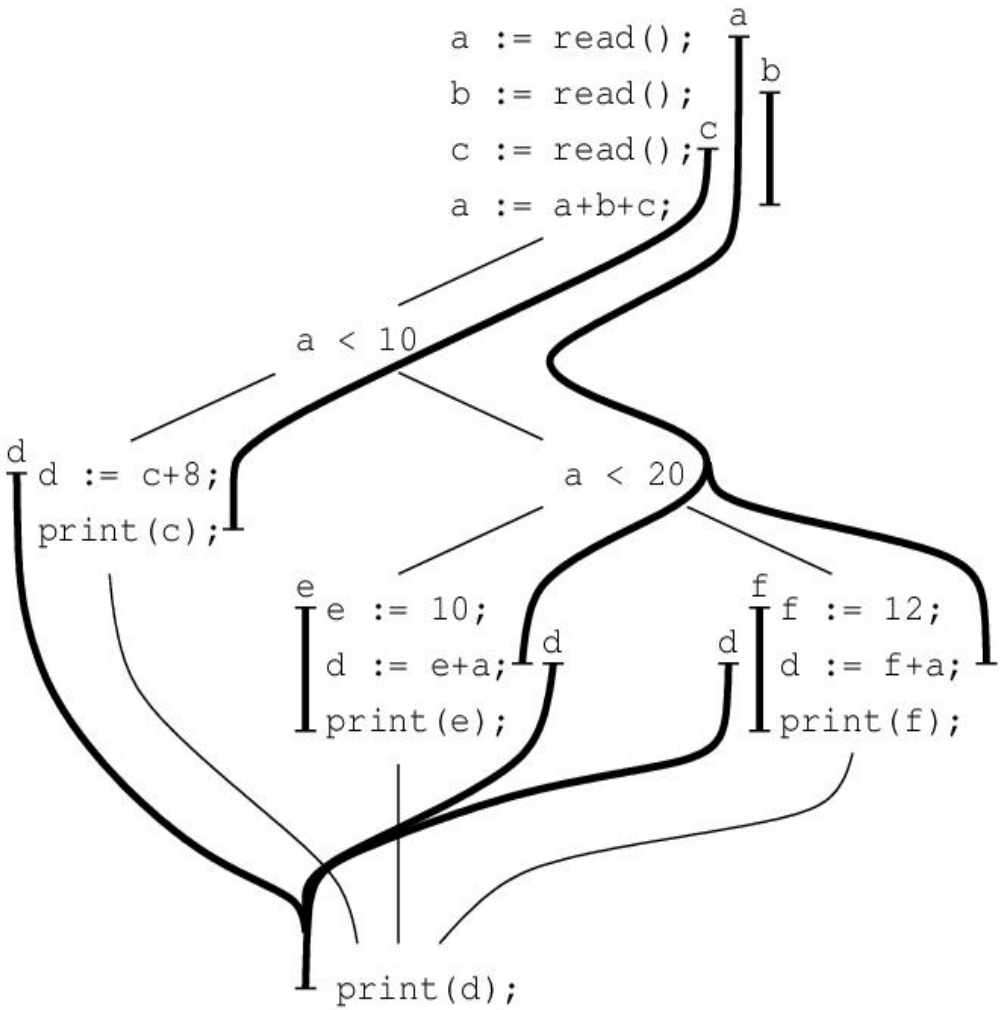
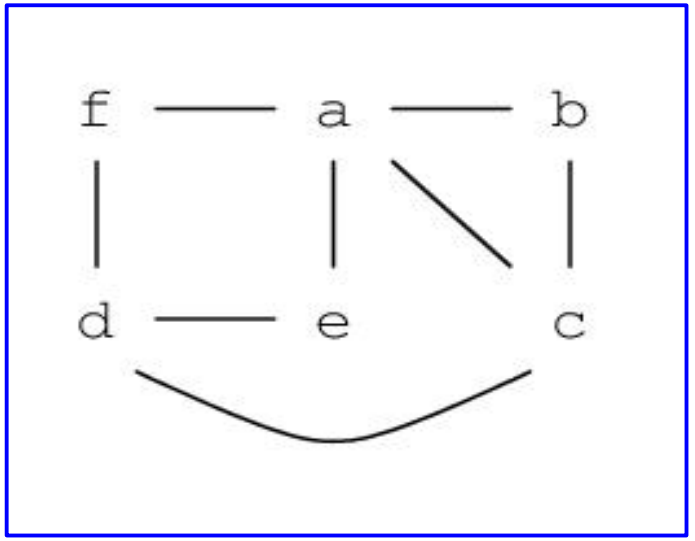


Figure 4.80 Live ranges of the variables from Figure 4.79.

Register-Interferenz-graph (Fig. 4.81):



Heuristik zur Graphfärbung

- exaktes Verfahren zur Graphfärbung: NP-hart
  - rechenzeitaufwändig
  - für jede Routine durchzuführen
- Alternative: Heuristik

1. spalte einen Knoten N mit minimalem(!) Grad k vom Graphen ab
2. merke Dir die Nachbarn $M_1,...,M_k$ von N
3. verfahre rekursiv mit dem verbleibenden Graphen
   $\Rightarrow$ eine Färbung dieses Graphen mit der Farbenmenge C
4. färbe den abgespalteten Knoten:
   (a) falls $k < |C|$ oder zwei Knoten in $M_1,...,M_k$ haben die gleiche Farbe: mit einer Farbe aus C
   (b) sonst: mit einer neuen Farbe F, setze $C' := C \cup \{F\}$

Graphfärbung, Programm (1)

- suche einen Knoten kleinsten Grades:

```
FUNCTION Color graph (Graph) RETURNING the Colors used set:
    IF Graph = the Empty graph: RETURN the Empty set;

    // Find the Least connected node:
    SET Least connected node TO No node;
    FOR EACH Node IN Graph .nodes:
        SET Degree TO 0;
        FOR EACH Arc IN Graph .arcs:
            IF Arc contains Node:
                Increment Degree;
        IF Least connected node = No node OR
            Degree < Minimum degree:
            SET Least connected node TO Node;
            SET Minimum degree TO Degree;
```

Graphfärbung, Programm (2)

- entferne gefundenen Knoten
- rekursiver Aufruf

```
// Remove Least connected node from Graph:
SET Least connected node arc set TO the Empty set;
FOR EACH Arc IN Graph .arcs:
    IF Arc contains Least connected node:
        Remove Arc from Graph .arcs;
        Insert Arc in Least connected node arc set;
Remove Least connected node from Graph .nodes;

// Color the reduced Graph recursively:
SET Colors used set TO Color graph (Graph);
```

Graphfärbung, Programm (3)

- färbe den vor dem rekursiven Aufruf entfernten Knoten

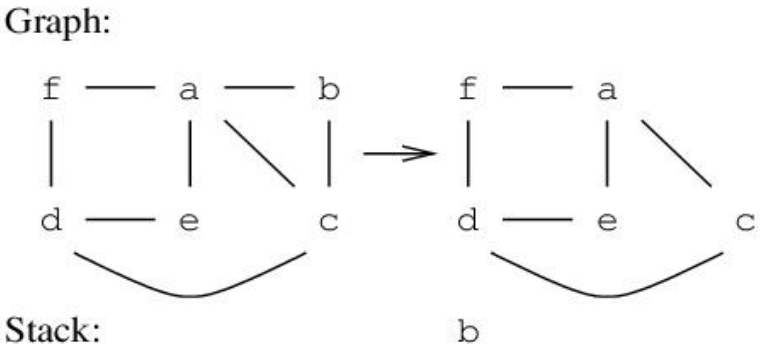
```
// Color the Least connected node:
SET Left over colors set TO Colors used set;
FOR EACH Arc IN Least connected node arc set:
    FOR EACH End point IN Arc:
        IF End point /= Least connected node:
            Remove End point .color from Left over colors set;
IF Left over colors set = Empty:
        SET Color TO New color;
        Insert Color in Colors used set;
        Insert Color in Left over colors set;
SET Least connected node .color TO
    Arbitrary choice from Left over colors set;
```

Graphfärbung, Programm (4)

- füge den entfernten Knoten wieder zum Graphen hinzu

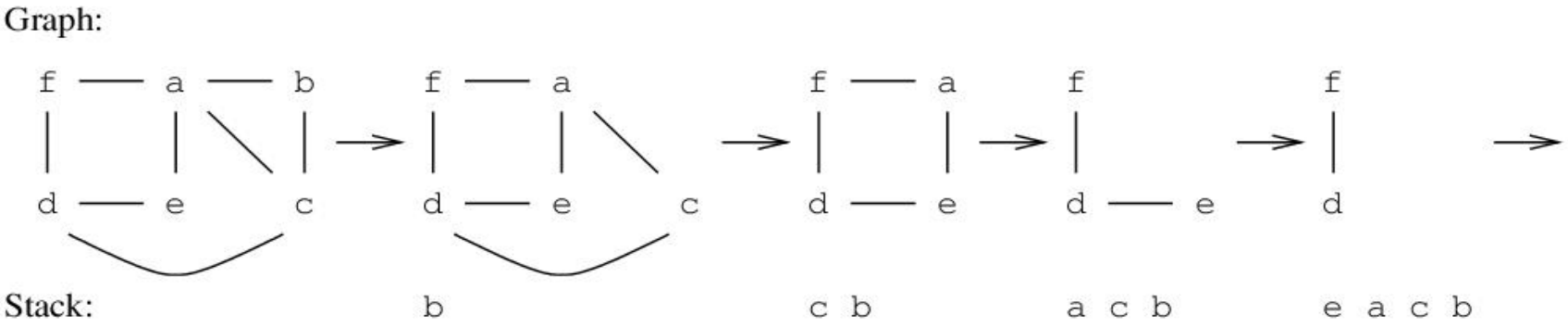
```
// Reattach the Least connected node:
Insert Least connected node in Graph .nodes;
FOR EACH Arc IN Least connected node arc set:
    Insert Arc in Graph .arcs;
RETURN Colors used set;
```

Graphfärbung, Beispiel (1)



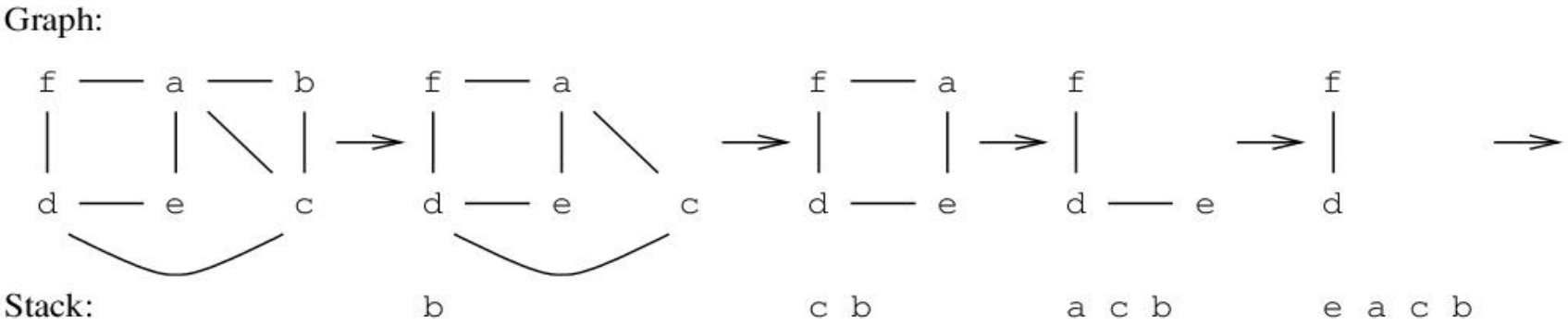
Abspaltung von b

Graphfärbung, Beispiel (2)



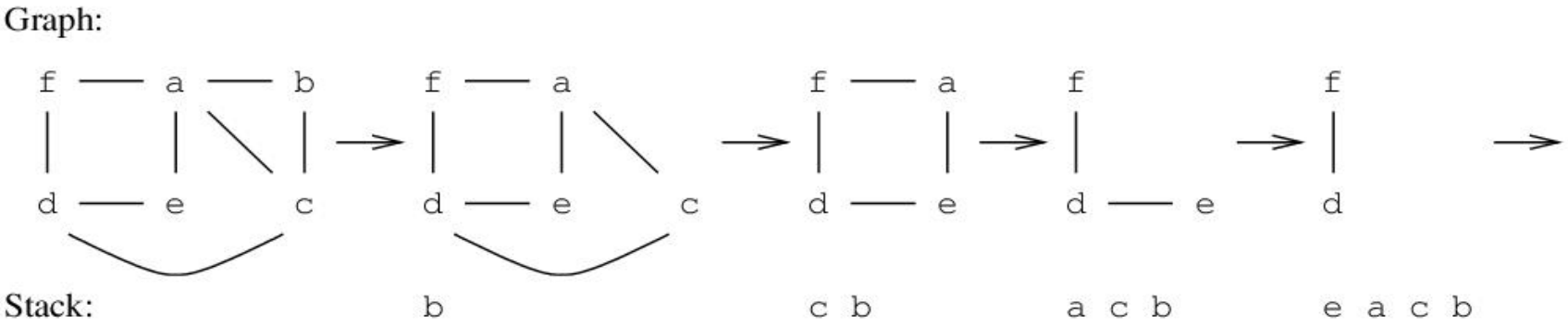
Abspaltung von b, c, a und e

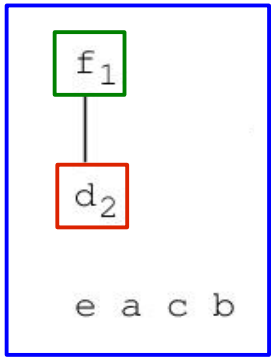
Graphfärbung, Beispiel (3)



- Abspaltung von `f`
- Graph leer
- Hinzunahme von `f`
- Färbung von `f` mit Farbe 1

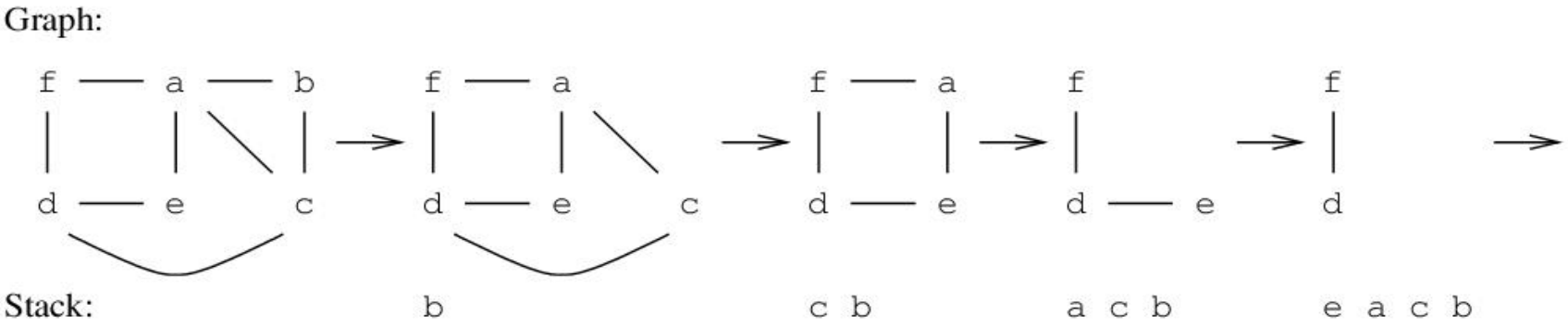
Graphfärbung, Beispiel (4)

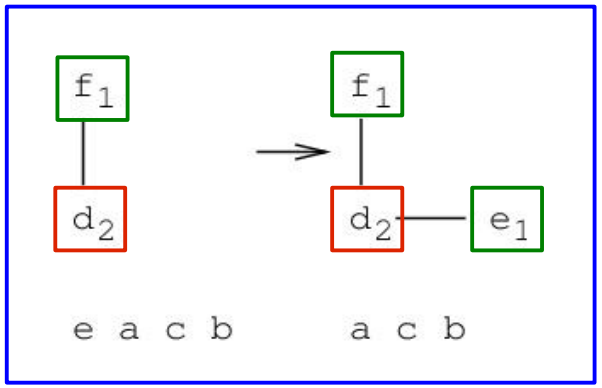




- Hinzunahme von d
- Kante (Konflikt) mit f
- Färbung von d mit Farbe 2

Graphfärbung, Beispiel (5)





- Hinzunahme von e
- Konflikt mit `d`, aber nicht mit `f`
- gleiche Färbung wie `f`

Graphfärbung, Beispiel (6)

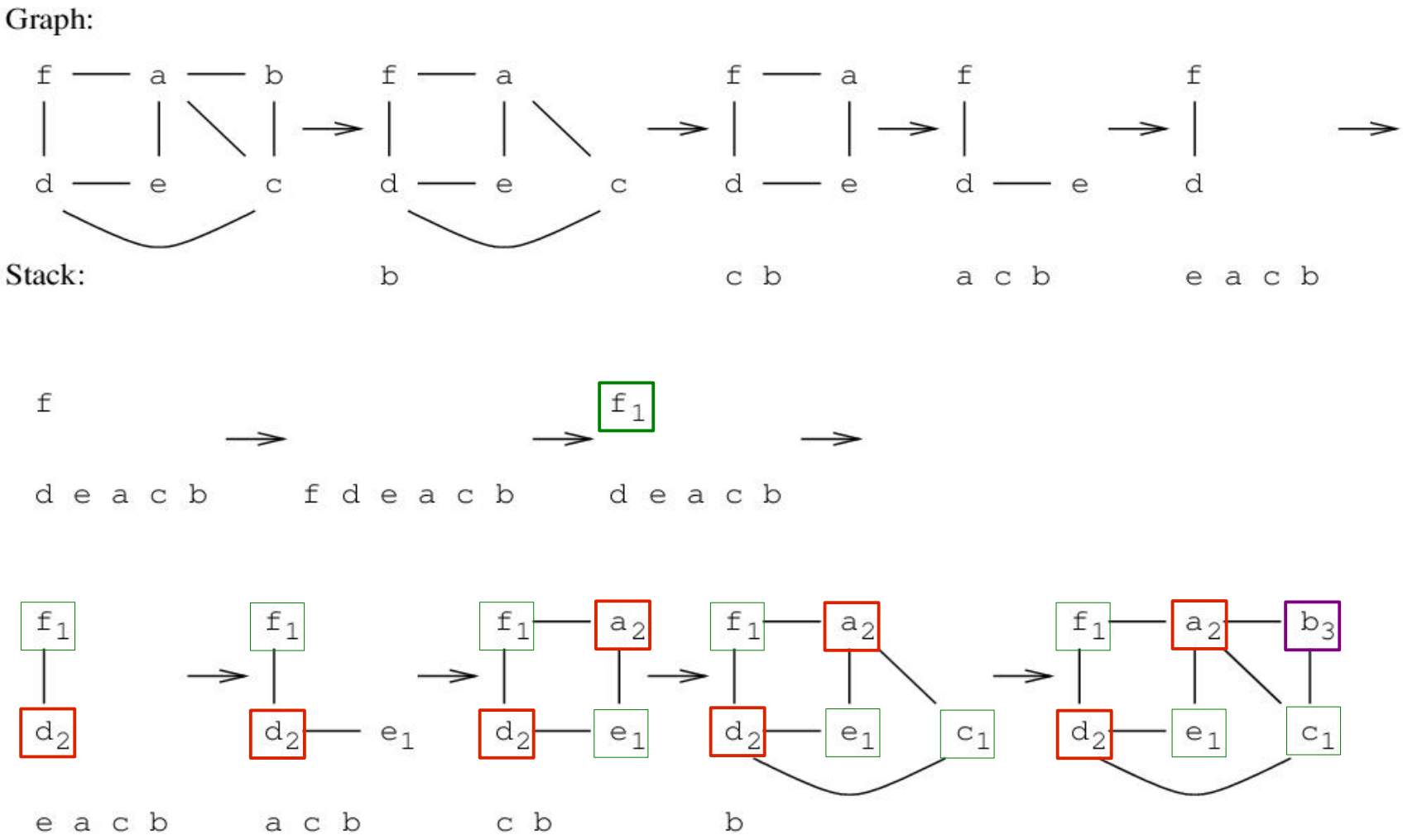


Figure 4.83 Coloring the interference graph for the variables of Figure 4.79.

Usage-Count-Methode

- Vorgehen bei Registermangel
  - jede Farbe entspricht jetzt zunächst einem Pseudoregister
- “zähle” bei jedem Pseudoregister die Häufigkeit des Vorkommens
  - statisch: Summe der mit Wahrscheinlichkeiten gewichteten Zugriffe
  - dynamisch: Profiling mit (realistischen) Testeingaben
- sortiere Pseudoregister nach absteigender Häufigkeit
  - bilde die ersten Pseudoregister auf verfügbare Register ab
  - restliche Pseudoregister werden zu Speicherstellen

Supercompilation (1)

- Einsatz fest vorgegebener Folgen von Maschinenbefehlen (für häufig vorkommende Funktionen)
- Finden einer optimalen Folge durch Simulation aller sinnvollen Kombinationen (in aufsteigender Ausführungszeit)
  - Berücksichtigung von Bedingungsregistern
  - Einschränkung auf Registeroperationen, keine Sprünge
- Vorauswahl durch Tests
  - z.B. 1000 Testfälle pro Kombination
  - im Negativfall oft schon Abbruch nach wenigen Tests
  - im Positivfall: manuelle Verifikation der Kombination nötig
- geschätzter Zeitaufwand für das Finden der optimalen Sequenz
  - mit ein paar Befehlen: einige Stunden
  - mit ca. 12 Befehlen: ein paar Wochen

Supercompilation (2)

Beispiel: optimale Codesequenz für `sign`-Funktion

```
sign(n) = -1, falls n<0
sign(0) = 0
sign(n) = 1, falls n>0
```

```
; n in register %ax
    cwd              ; convert to double word:
                     ;     (%dx,%ax) = (extend_sign(%ax), %ax)
    negw %ax         ; negate: (%ax,cf) := (-%ax, %ax /= 0)
    adcw %dx,%dx     ; add with carry: %dx := %dx + %dx + cf
; sign(n) in %dx
```

Figure 4.84 Optimal code for the function `sign(n)`.

Supercompilation (3)

Verifikation mittels Fallunterscheidung

	Case n > 0			Case n = 0			Case n < 0		
	%dx	%ax	cf	%dx	%ax	cf	%dx	%ax	cf
initially:									
	–	n	–	–	0	–	–	n	–
cwd									
	0	n	–	0	0	–	-1	n	–
negw %ax									
	0	-n	1	0	0	0	-1	-n	1
adcw %dx,%dx									
	1	-n	1	0	0	0	-1	-n	1

Figure 4.85 Actions of the 80x86 code from Figure 4.84.

Codegenerierungstechniken im Vergleich

Problem	Technik	Güte
Ausdrucksbäume (Reg/Reg, Reg/Mem)	Gewichtete Bäume	optimal (unter den Einschränkungen)
Abhängigkeitsgraphen (Reg/Reg, Reg/Mem)	Leitersequenzen	heuristisch
Ausdrucksbäume mit Befehlen mit Kostenfkt.	Bottom-Up Tree Rewriting (BURS)	genügend Register: optimal, sonst: heuristisch
Registerbelegung für Datenflussgraphen, bei bekannten Konflikten	Graphfärbung	exakt: aufwändig (NP-hart) heuristisch: schnell

Debugging von Codeoptimierern (1)

- real existierende Optimierer
  - hunderte Techniken und Tricks
  - viele Spezialfälle, kompliziert implementiert
  - viele potenzielle Fehlerquellen, u.U. erst durch Kombination
- Einfluss auf Programmkorrektheit
  - Optimierung beseitigt (maskiert) Fehler, denkbar
  - Fehler tritt erst bei Optimierung auf ⇒ liegt der Fehler im Programm oder beim Optimierer?
- Vorgehensweise
  - Test mit/ohne eine bestimmte Optimierung
  - Fehlereingrenzung: Test mit der einen/anderen Hälfte der Compileroptionen
  - Vermeidung aller Optimierungen, wo ein Fehler sichtbar wird

Debugging von Codeoptimierern (2)

```
int i, A[10];

for (i = 0; i < 20; i++) {
    A[i] = 2*i;
}
```

Figure 4.87 Incorrect C program with compilation-dependent effect.

- Programmfehler(!): das Array hat nur `10` Elemente, in der Schleife werden aber die Indizes `10..19` verwendet
- nicht-optimiertes Programm arbeitet scheinbar fehlerfrei
  - `i` liegt im Speicher direkt hinter `A[9]`, `i` "*Alias*" für `A[10]`
  - `A[10]=2*10` setzt `i` auf `20`, Schleife terminiert
  - erster Fehler führt dazu, dass weiterer fehlerhafter Code nicht mehr ausgeführt wird

Debugging von Codeoptimierern (3)

```
int i, A[10];

for (i = 0; i < 20; i++) {
    A[i] = 2*i;
}
```

Figure 4.87 Incorrect C program with compilation-dependent effect.

- Programmfehler(!): das Array hat nur `10` Elemente, in der Schleife werden aber die Indizes `10..19` verwendet
- nicht-optimiertes Programm: Fehler zeigt sich nicht
- optimiertes Programm: Fehler wird sichtbar
  - `i` wird in einem Register gehalten
  - die Schleife schreibt ab `i=10` in einen nicht dafür vorgesehenen Speicherbereich Crash (o.ä.)