

# STUPS

## Code Generation 2: Code Generierung und Code Optimierung

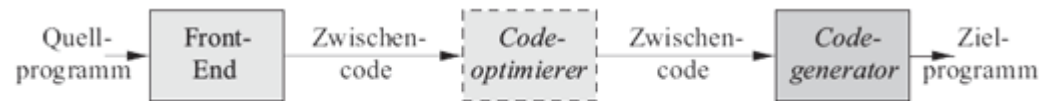


Abbildung 8.1: Die Position des Codegenerators

# Wiederholung: Format des Zwischencodes

- Befehle:
  - $x = y \text{ op } z$ ,  $x = \text{op } y$ ,  $x = y$
  - goto L
  - if x goto L und ifFalse x goto L
  - if x relop y goto L
  - $x = y[i]$  und  $x[i] = y$
  - $x = \&y$ ,  $x = *y$ ,  $x* = y$
- Adressen:
  - Namen, Konstanten, temporäre Variablen

Wiederholung:  
Drei-Adress RISC Maschine

- LD dst, addr (LD r,x oder LD r1,r2)
- ST x,r
- OP dst, src1, src2 (SUB r1,r2,r3:  $r1=r2-r3$ )
- BR lbl
- Bcond r, L
- Adressen:
  - x : Variablennamen
  - a(reg): a Variablenname
  - cst(reg), \*r, \*cst(r)
  - #cst (LD r1, #100)

dies ist die  
Zielsprache

# Speicherverwaltung

*(ganz kurze Exkursion;  
komplette Details in Kapitel 7)*



Abbildung 7.1: Gewöhnliche Unterteilung des Laufzeitspeichers in Code- und Datenbereiche

# Speicherverwaltung II

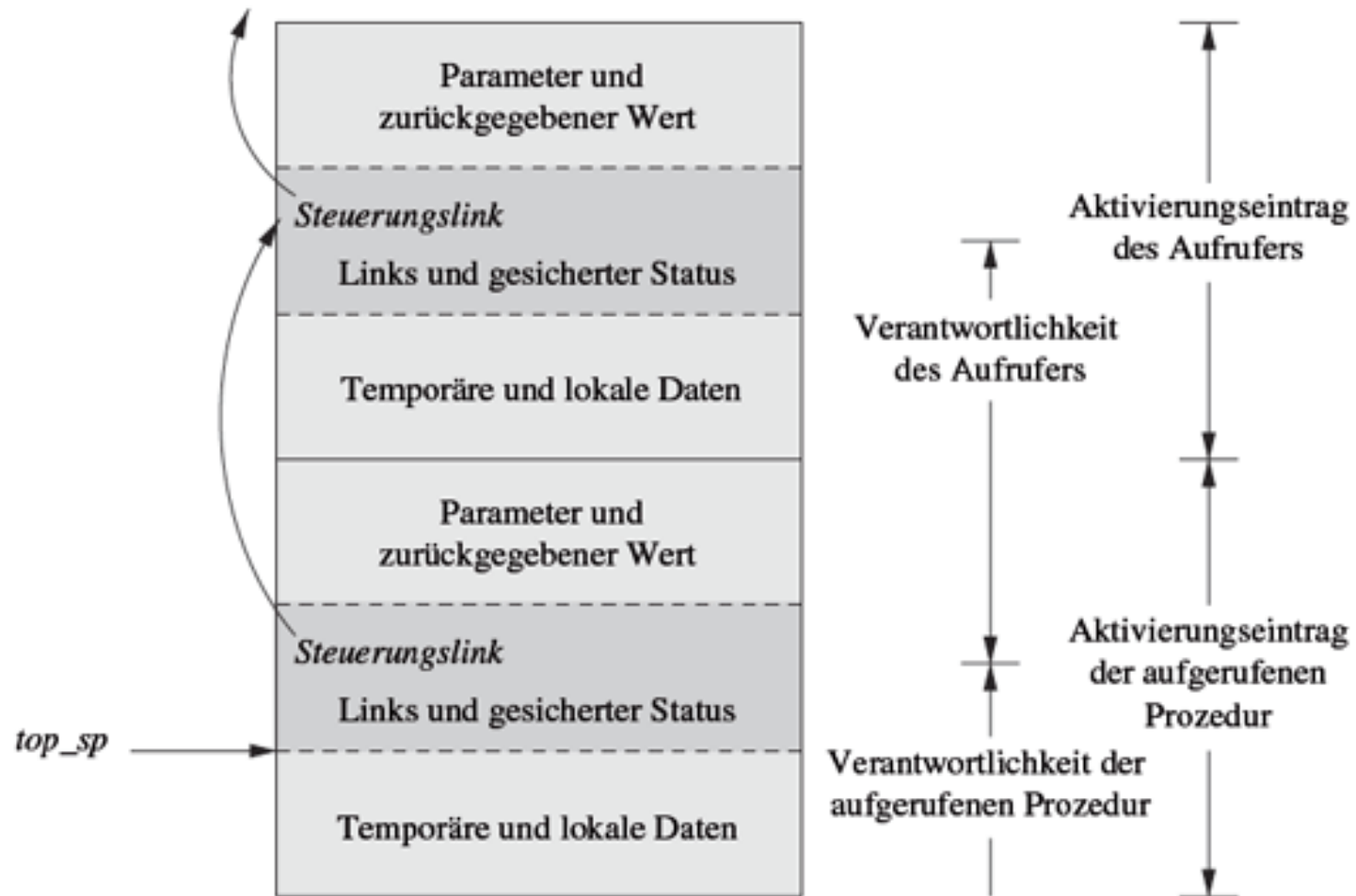


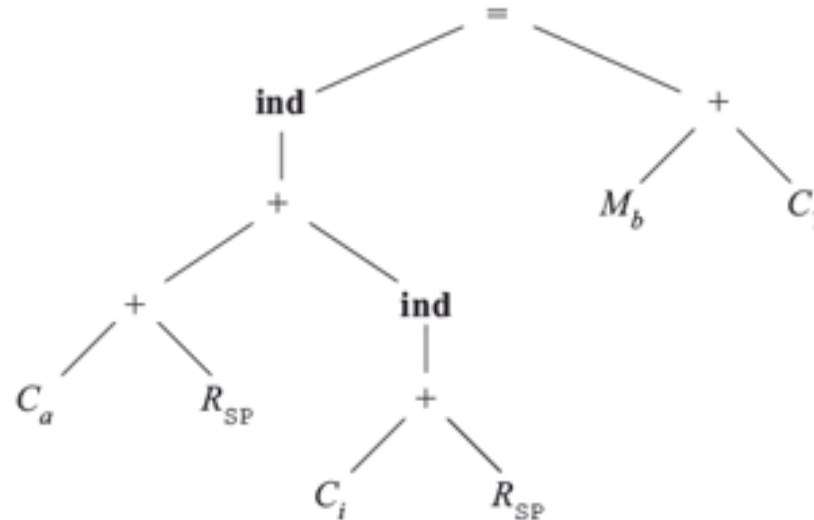
Abbildung 7.7: Aufgabenteilung zwischen Aufrufer und aufgerufener Prozedur

# Instruction Selection (Befehlsauswahl)

- Generating real machine instructions from IR tree (or intermediate code)
- Followed by register allocation
- Not a 1:1 correspondence between tree nodes and instructions, e.g.:

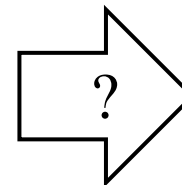
# Zwischencode nach Zielcode

- Zwischencode (nach Adresszuweisung) in Baumdarstellung:



Ca, Ci: Offsets im Stackframe  
Mb: Adresse der globalen Variable b  
ind: Indirektion

Abbildung 8.19: Zwischencodebaum für  $a[i] = b + 1$



```
LD R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD R1, b
INC R1
ST *R0, R1
```

Siehe Kapitel 8.3

# Baumersetzungsverfahren

“Kontextfreie Grammatik” auf Bäumen:



*Semantische  
Aktion*

*Anmerkung:  $i, j$  Parameter der Regel*

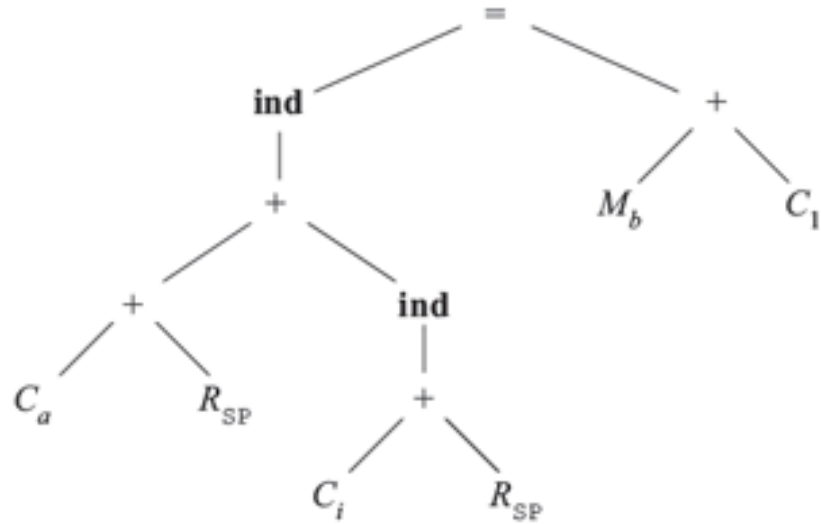
*Generierung des Maschinencodes durch LR-Parsing*

# Komplette Regeln

1)	$R_i \leftarrow C_a$	{ LD $R_i, \#a$ }
2)	$R_i \leftarrow M_x$	{ LD $R_i, x$ }
3)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ M_x \quad R_i \end{array}$	{ ST $x, R_i$ }
4)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ \mathbf{ind} \quad R_j \\   \\ R_i \end{array}$	{ ST $*R_i, R_j$ }
5)	$R_i \leftarrow \begin{array}{c} \mathbf{ind} \\   \\ + \\ / \quad \backslash \\ C_a \quad R_j \end{array}$	{ LD $R_i, a(R_j)$ }
6)	$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad \mathbf{ind} \\ \quad \quad   \\ \quad \quad + \\ \quad \quad / \quad \backslash \\ \quad \quad C_a \quad R_j \end{array}$	{ ADD $R_i, R_i, a(R_j)$ }
7)	$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad R_j \end{array}$	{ ADD $R_i, R_i, R_j$ }
8)	$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad C_1 \end{array}$	{ INC $R_i$ }

Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

# Beispiel



1)  $R_0 \leftarrow C_a$  { LD R0, #a }

7)  $R_0 \leftarrow$   { ADD R0, R0, SP }

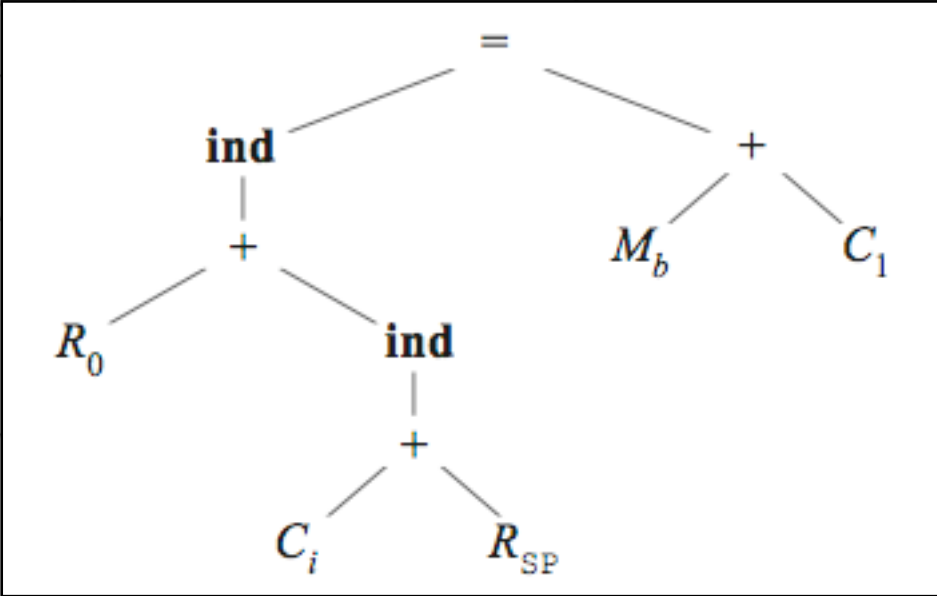
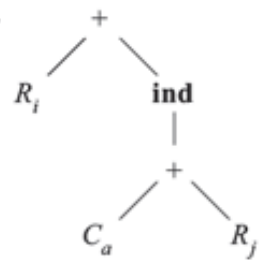
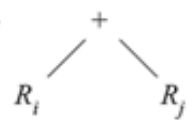
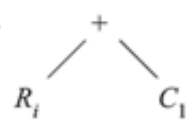
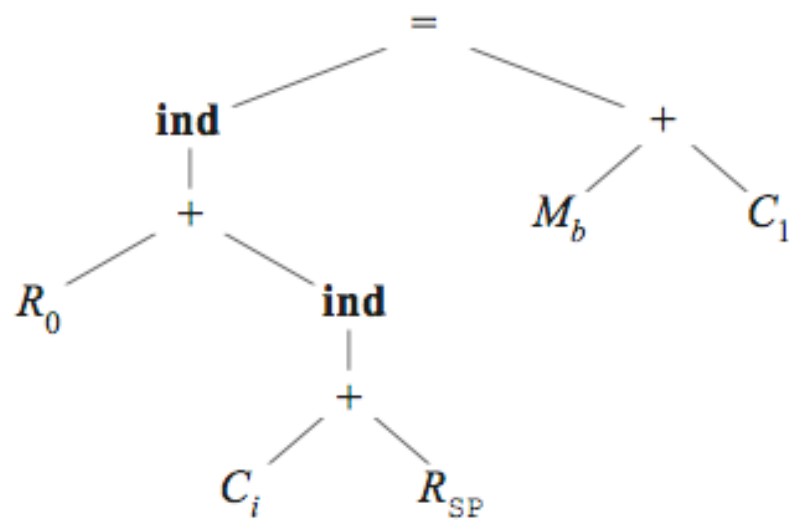
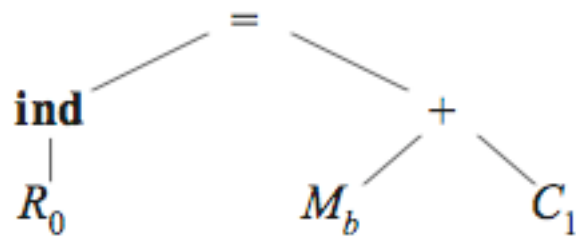
1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
		
6)	$R_i \leftarrow$ 	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \leftarrow$ 	{ ADD Ri, Ri, Rj }
8)	$R_i \leftarrow$ 	{ INC Ri }

Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

# Beispiel



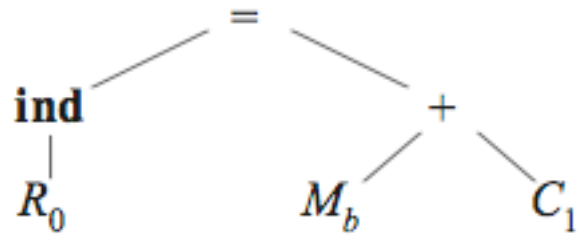
ADD R0, R0, i(SP)



1)	$R_i \leftarrow C_a$	{ LD $R_i, \#a$ }
2)	$R_i \leftarrow M_x$	{ LD $R_i, x$ }
3)	$M \leftarrow =$ $\begin{array}{cc} M_x & R_i \end{array}$	{ ST $x, R_i$ }
4)	$M \leftarrow =$ $\begin{array}{cc} \text{ind} & R_j \\   & \\ R_i & \end{array}$	{ ST $*R_i, R_j$ }
5)	$R_i \leftarrow \text{ind}$ $\begin{array}{cc} + \\ / \quad \backslash \\ C_a \quad R_j \end{array}$	{ LD $R_i, a(R_j)$ }
6)	$R_i \leftarrow +$ $\begin{array}{cc} R_i & \text{ind} \\ &   \\ & + \\ & / \quad \backslash \\ & C_a \quad R_j \end{array}$	{ ADD $R_i, R_i, a(R_j)$ }
7)	$R_i \leftarrow +$ $\begin{array}{cc} R_i & R_j \end{array}$	{ ADD $R_i, R_i, R_j$ }
8)	$R_i \leftarrow +$ $\begin{array}{cc} R_i & C_1 \end{array}$	{ INC $R_i$ }

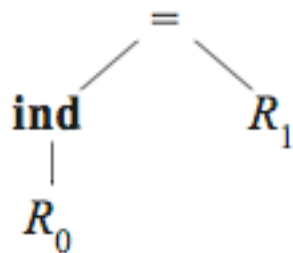
Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

# Beispiel



Regel 2    LD R0, #a  
           ADD R0, R0, SP  
           ADD R0, R0, i(SP)  
           LD R1, b

Regel 8 (INC R1)



Regel 4:    INC R1  
           ST \*R0, R1

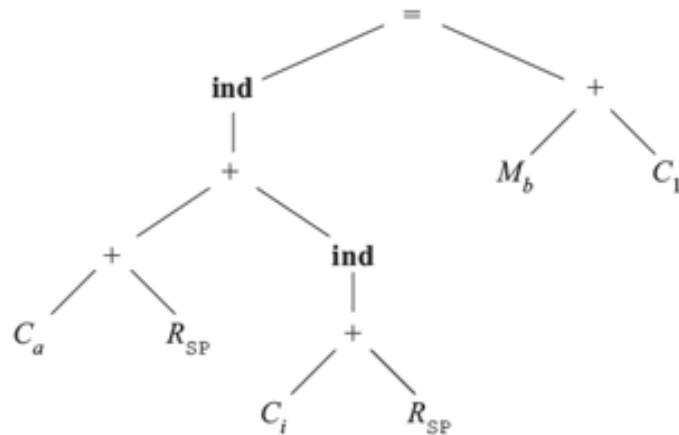
1)	$R_i \leftarrow C_a$	{ LD $R_i, \#a$ }
2)	$R_i \leftarrow M_x$	{ LD $R_i, x$ }
3)	$M \leftarrow =$ $\begin{array}{cc} M_x & R_i \end{array}$	{ ST $x, R_i$ }
4)	$M \leftarrow =$ $\begin{array}{cc} \text{ind} & R_j \\   & \\ R_i & \end{array}$	{ ST $*R_i, R_j$ }
5)	$R_i \leftarrow \text{ind}$ $\begin{array}{cc} + \\ / \quad \backslash \\ C_a \quad R_j \end{array}$	{ LD $R_i, a(R_j)$ }
6)	$R_i \leftarrow +$ $\begin{array}{cc} R_i & \text{ind} \\ &   \\ & + \\ & / \quad \backslash \\ & C_a \quad R_j \end{array}$	{ ADD $R_i, R_i, a(R_j)$ }
7)	$R_i \leftarrow +$ $\begin{array}{cc} R_i & R_j \end{array}$	{ ADD $R_i, R_i, R_j$ }
8)	$R_i \leftarrow +$ $\begin{array}{cc} R_i & C_1 \end{array}$	{ INC $R_i$ }

Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

# Mustererkennung auf Bäumen mit LR-Parsing

andere Ansätze in 8.9.5 oder in Appel

- Baum  $\rightarrow$  String der Präfixdarstellung



$\rightarrow$

= ind + + C<sub>a</sub> R<sub>SP</sub> ind + C<sub>i</sub> R<sub>SP</sub> + M<sub>b</sub> C<sub>1</sub>

Zwischencodebaum für  $a[i] = b + 1$

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ M_x \quad R_i \end{array}$	{ ST x, Ri }
4)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ \text{ind} \quad R_j \\   \\ R_i \end{array}$	{ ST *Ri, Rj }
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\   \\ + \\ / \quad \backslash \\ C_a \quad R_j \end{array}$	{ LD Ri, a(Rj) }
6)	$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad \text{ind} \\ \quad \quad   \\ \quad \quad + \end{array}$	{ ADD Ri, Ri, a(Rj) }

$\rightarrow$

- 1)  $R_i \rightarrow c_a$  { LD Ri, #a }
- 2)  $R_i \rightarrow M_x$  { LD Ri, x }
- 3)  $M \rightarrow = M_x R_i$  { ST x, Ri }
- 4)  $M \rightarrow = \text{ind } R_i R_j$  { ST \*Ri, Rj }
- 5)  $R_i \rightarrow \text{ind } + c_a R_j$  { LD Ri, a(Rj) }
- 6)  $R_i \rightarrow + R_i \text{ind } + c_a R_j$  { ADD Ri, Ri, a(Rj) }
- 7)  $R_i \rightarrow + R_i R_j$  { ADD Ri, Ri, Rj }
- 8)  $R_i \rightarrow + R_i c_1$  { INC Ri }
- 9)  $R \rightarrow \text{sp}$
- 10)  $M \rightarrow m$

Abbildung 8.21: Aus Abbildung 8.20 konstruiertes syntaxgesteuertes Übersetzungsverfahren

# Optimal Tilings

- “Best” tiling → least cost instruction sequence
  - smallest number of instructions
  - lowest execution time
- *Optimum* tiling: minimizes sum of costs of tiles
- *Optimal* tiling: cannot combine adjacent tiles into a tile of lower cost
- CISC processors: large tiles, variable cost
  - optimum may be significantly better than optimal
- RISC processors: small tiles, uniform cost
  - not much difference

# An Optimal Tiling Algorithm

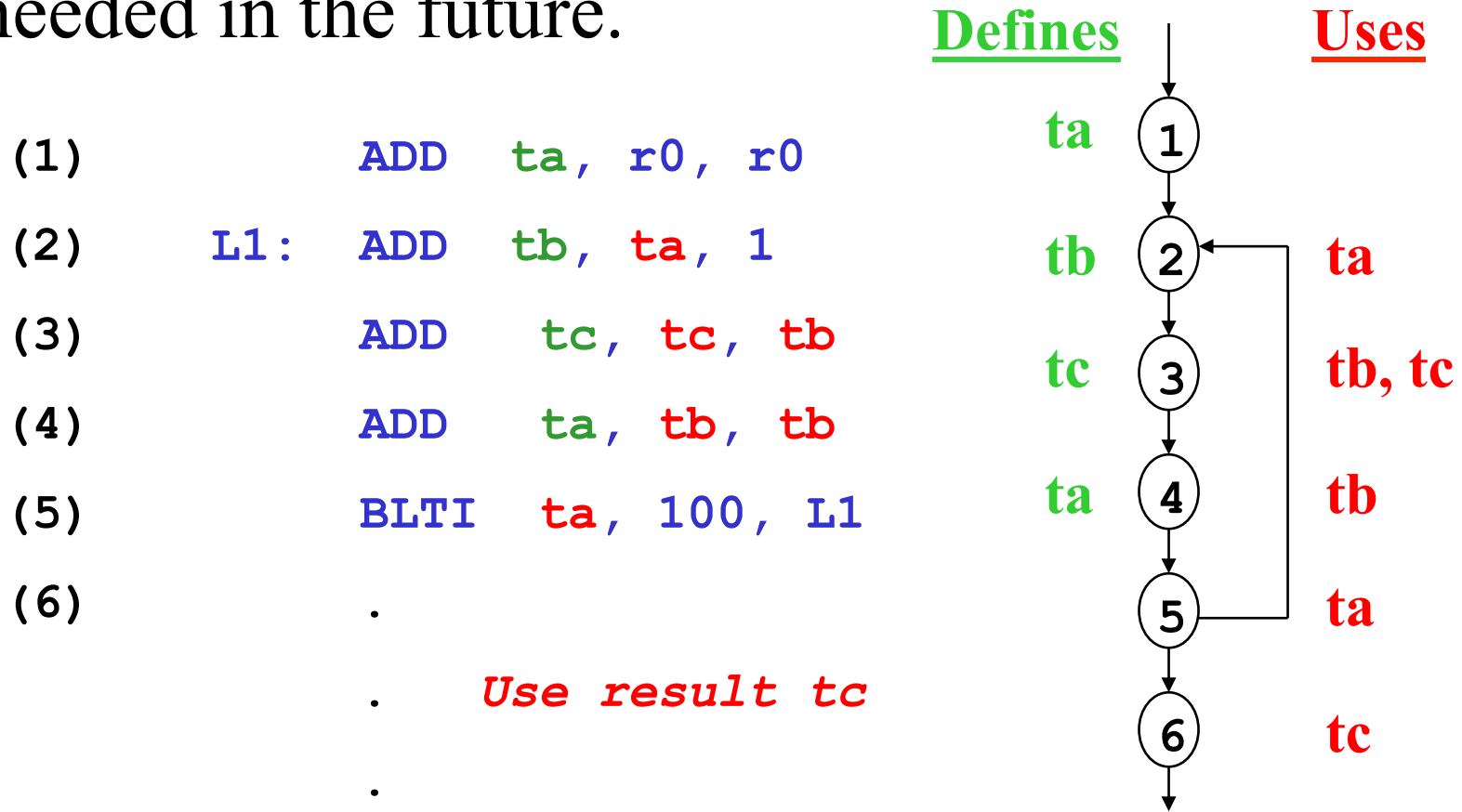
- “Maximal Munch”
  - Find largest tile that fits
  - Use to cover that part of tree, leaving several subtrees
  - Repeat for each subtree
- “Largest” = covers the most nodes.
- Arbitrary choice where more than one.
  
- NB: Order in which instructions are emitted is different.

# Register Allocation

- Result of instruction selection still uses (mainly) “indeterminate” registers
- Need to allocate to real registers
- Allocate multiple temporaries to same real register
  - when not “in use” at the same time
- Spill into memory if necessary
  - involves modifying instruction sequence

# Liveness Analysis

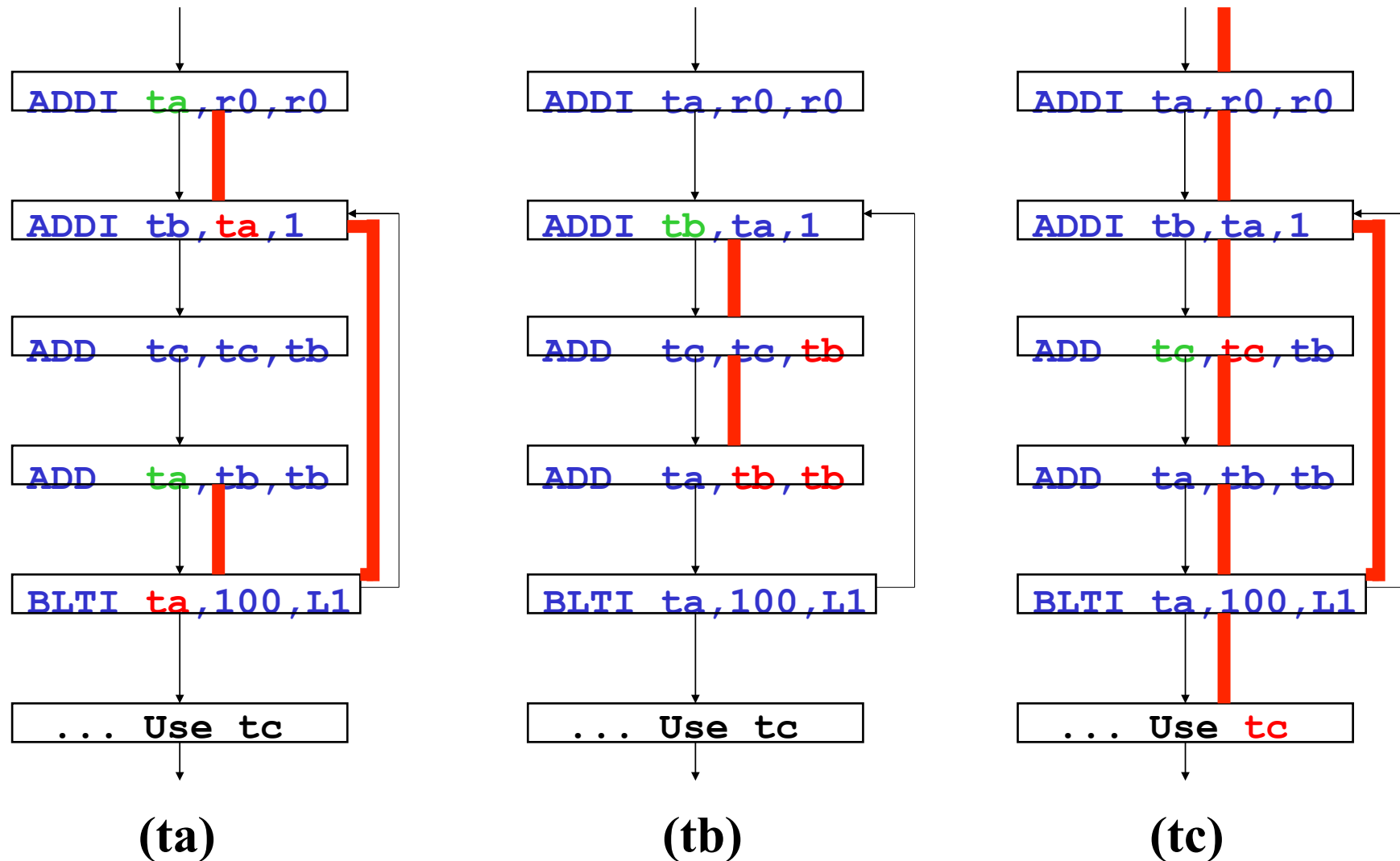
- A variable is **live** if it holds a value that may be needed in the future.



Control Flow Graph

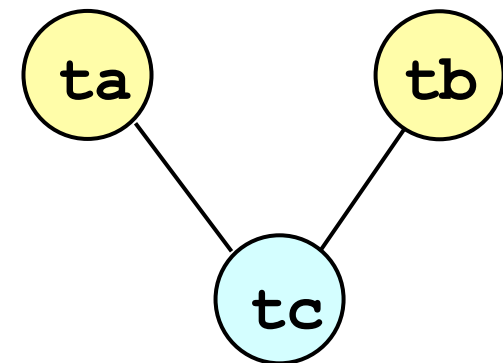
# Liveness Analysis (2)

- **Liveness** of *ta*, *tb* and *tc*:



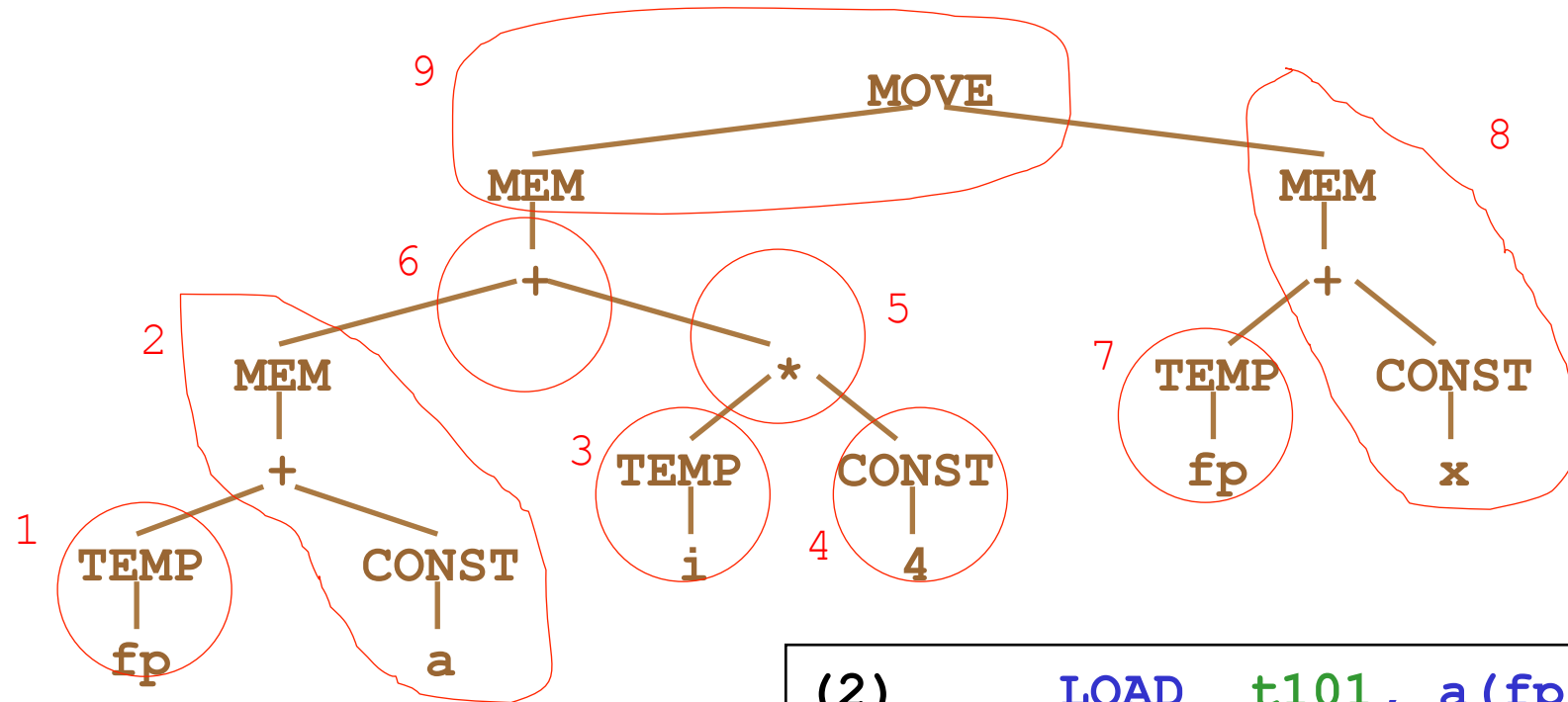
# Interference Graph

- **ta** and **tb** are never live simultaneously, so can share a register.
- Express constraints as *interference graph*
  - nodes represent temporaries
  - two nodes connected by an edge if they cannot be allocated to the same register
- Use graph colouring algorithm to allocate available registers to nodes.



Interference graph for previous program:

# Example Tiling (from Appel Book)

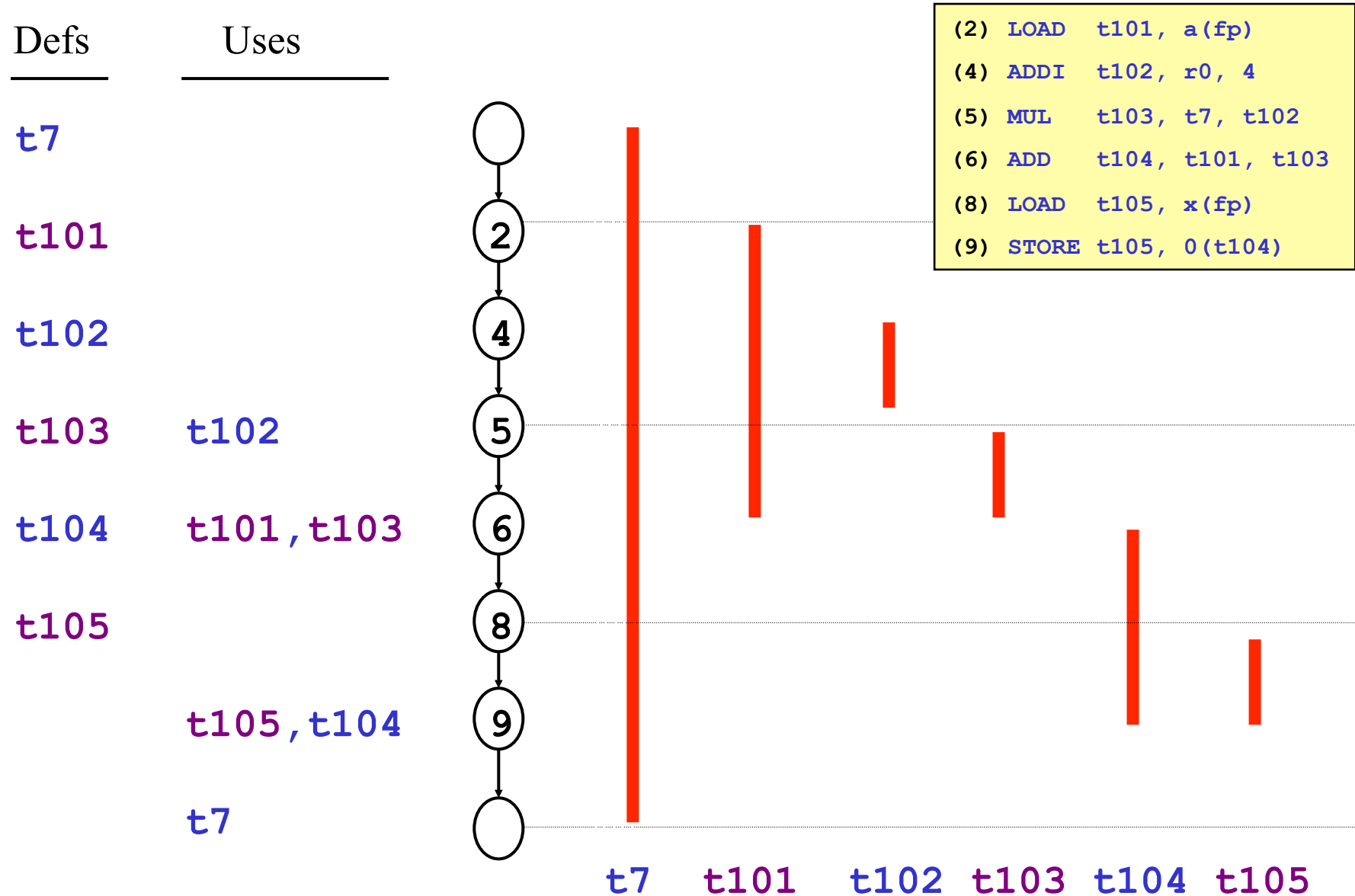


Code actually generated  
by instruction selection  
phase.

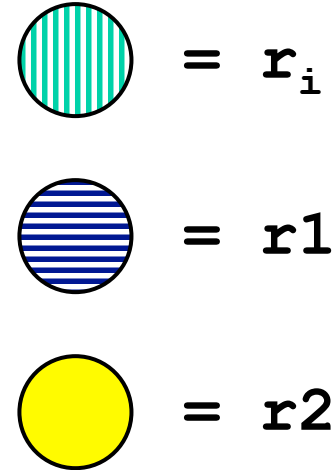
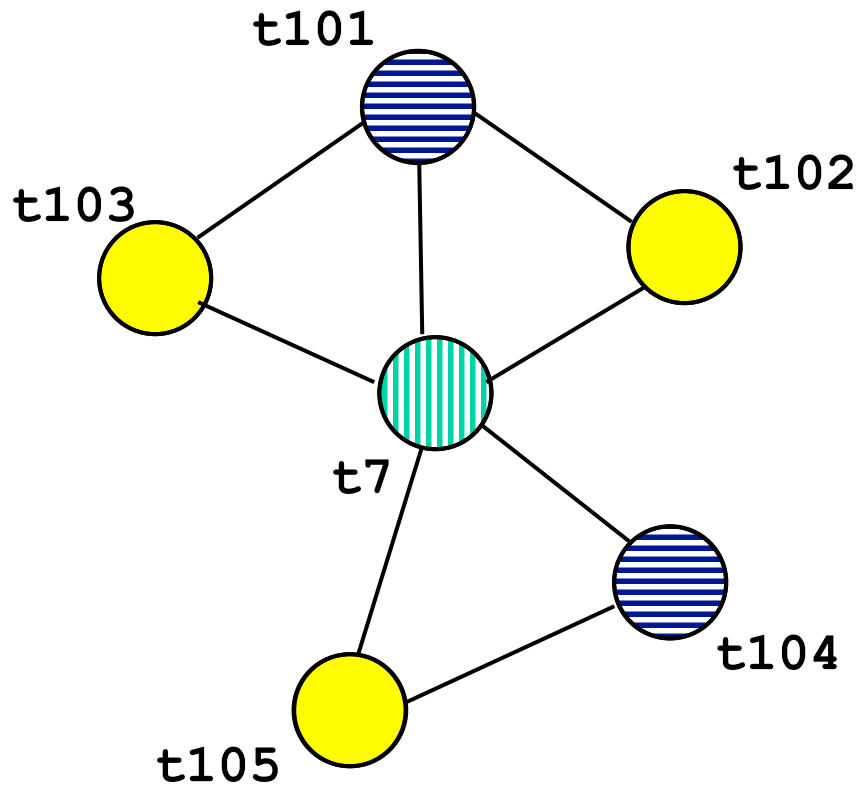
Variable **i** allocated to **t7**.

(2)	LOAD	t101, a(fp)
(4)	ADDI	t102, r0, 4
(5)	MUL	t103, t7, t102
(6)	ADD	t104, t101, t103
(8)	LOAD	t105, x(fp)
(9)	STORE	t105, 0(t104)

# Example: Liveness Analysis



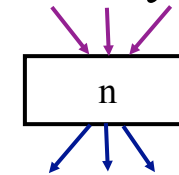
# Example: Interference Graph



```
LOAD  r1, a(fp)
ADDI  r2, r0, 4
MUL   r2, ri, r2
ADD   r1, r1, r2
LOAD  r2, x(fp)
STORE r2, 0(r1)
```

# Computing Liveness

- **Def**(n) = set of vars defined in node n
- **Use**(n) = set of vars used
- Var **Live** on edge: directed path from that edge to use not via any def
- **in**[n] = set of vars live on any in-edge (**live-in**)
- **out**[n] = set of vars live on any out-edge (**live-out**)



$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

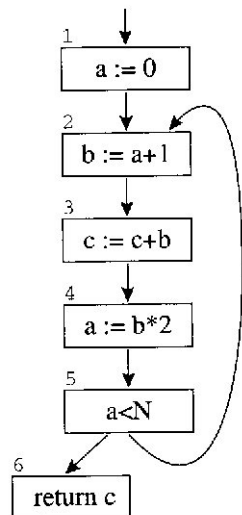
---

**EQUATIONS 10.3.** Dataflow equations for liveness analysis.

---

```

a ← 0
L1 : b ← a + 1
      c ← c + b
      a ← b * 2
      if a < N goto L1
      return c
    
```



```

for each n
  in[n] ← {}; out[n] ← {}
repeat
  for each n
    in'[n] ← in[n]; out'[n] ← out[n]
    in[n] ← use[n] ∪ (out[n] - def[n])
    out[n] ← ∪s ∈ succ[n] in[s]
  until in'[n] = in[n] and out'[n] = out[n] for all n
    
```

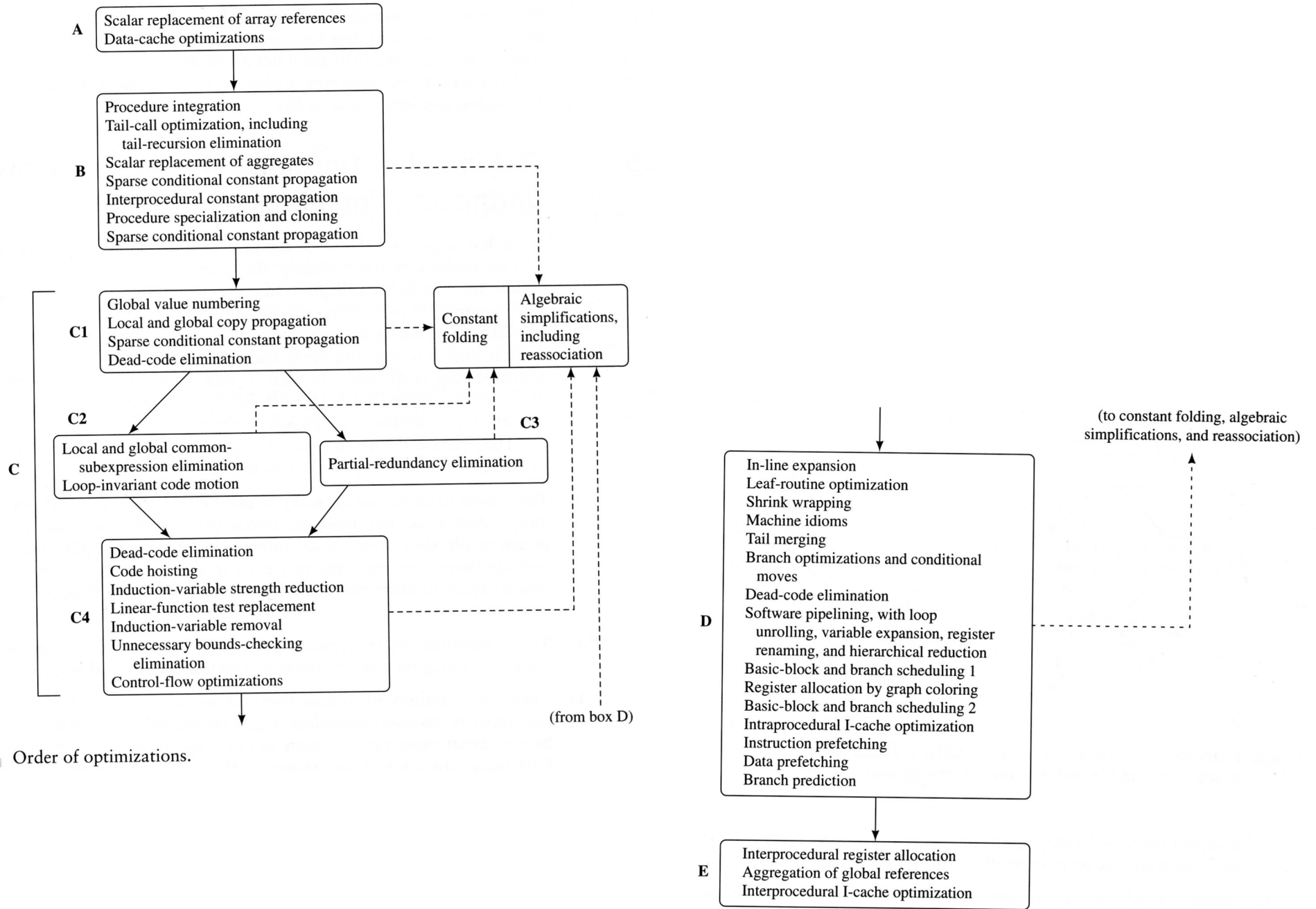
---

**ALGORITHM 10.4.** Computation of liveness by iteration.

---

# Code Optimization

- Techniques for improving quality of generated code.
- Law of diminishing returns applies.
- Most important is good register allocation.
- *Peephole optimization* applied locally to generated target code: remove redundant code:
  - `MOV R1,R1` (or `ADD r1,r0,r1`)
  - `JUMP L1`
  - `L1: ...`



(continued)

# Common Subexpressions

- Eliminate repeated evaluation of expressions whose value does not change.

- E.g.

`a[i+j] = a[i+j] + 1;`

calculates

`(base address of a) + (i + j)*(element size)`

twice; save in a register and re-use.

- Usually done by transformations on AST  
(or IR tree)

# Constant Folding

- Evaluate expressions as much as possible at compile time.
- Obvious:  $2 + 5$
- Less obvious:  $((2 * x) + 1) * 3$
- Constant propagation:

```
double pi = 3.141592654;  
double twopi = 2*pi;
```

- Copy propagation:

```
a = y + z;
```

```
b = y;
```

```
c = b + z;  -- common subexpression
```

# Dead-Code Elimination

- Remove code that is unreachable, or has no effect.
- Common case:

```
#define DEBUG 0  
.  
.  
    if (DEBUG) { ... }
```

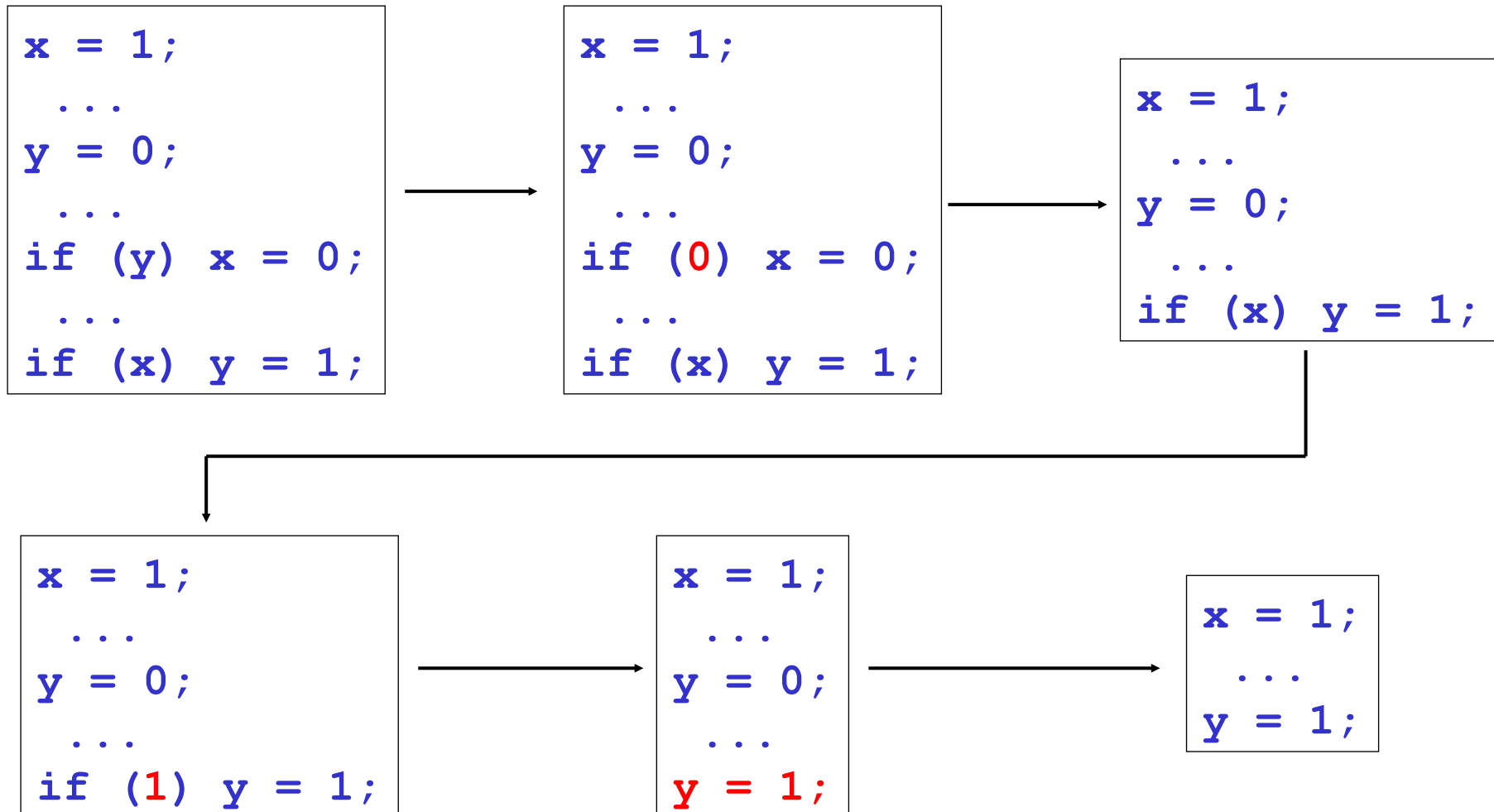
- But be careful:

```
.  
x = y / z;  
return y;
```

- Optimization must preserve program behaviour.

# Order of Optimization

- Order of applying transformations matters: may have to iterate.



# Zusammenfassung

*(Was ist wichtig für die Klausur)*

- Zwischencode
  - Null-Address + Drei-Address Code, IR Bäume
- Generierung von 0-, 3-Addresscode vom AST
- Grundblöcke, Kontrollflussgraph
- Instruction Selection\*
- Register Allokation
  - Liveness Analyse, Interferenzgraph
- Code Optimierung\*

\*: Grundlagen, keine Details

# Was haben wir gelernt

