INF5110 – Compiler Construction

Code generation

Spring 2016

Outline

Outline

Code generation[a]

[a] This section is based on slides from Stein Krogdahl, 2015.

- note: *code generation* so far: AST^+ to intermediate code
  - three address code
  - P-code
- $\Rightarrow$ *intermediate code generation*
- i.e., we are still not there . . .
- material here: based on the (old) *dragon book* [Aho et al., 1986]
- there is also a new edition [Aho et al., 2007]

Intro: code generation

- goal: translate TA-code (=3A-code) to machine language
- machine language/assembler:
  - even *more* restricted
  - 2 address code
- limited number of *registers*
- different *address modes* with different *costs* (registers vs. main memory)

Goals

- efficent code[a]
- small code size also desirable
- but first of all: correct code

[a] When not said otherwise: efficiency refers in the following to efficiency of the generated code. Fastness of compilation may be important as well (and same for the size of the compiler itself, as opposed to the size of the generated code). Obviously, there are trade-offs to be made.

Code "optimization"

- often conflicting goals
- code generation: *prime* arena for achieving *efficiency*
- *optimal* code: undecidable anyhow,
- even for many more clearly defined subproblems: *untractable*
- "optimization": interpreted as: *heuristics* to achieve "good code" (without hope for *optimal* code)
- due to importance of optimization at code generation
  - time to bring out the "heavy artillery"
  - so far: all techniques (parsing, lexing, even type checking) are computationally "easy"
  - at code generation/optmization: perhaps *invest* in agressive, computationally complex and rather advanced techniques
  - many different techniques used

Outline

2-address machine code used here

- "typical" op-codes, but not a instruction set of a *concrete* machine
- two address instructions
- Note: cf. 3-address-code interpmediate representation vs. 2-address machine code
  - machine code is not lower-level/closer to HW because it has one argument less than 3AC
  - it's just one illustrative choice
  - the new dragon book: uses 3-address-machine code (being more modern)
- 2 address machine code: closer to *CISC* architectures,
- *RICS* architectures rather use 3AC.
- translation task from IR to 3AC or 2AC: comparable challenge

Format

OP source dest

- note: *order* of arguments[1]
- restriction on *source* and *target*
  - register or memory cell
  - source: can additionally be a constant

```
ADD a b   // b := a + b
SUB a b   // b := b - a
MUL a b   // b := b + a
GOTO i    // unconditional jump
```

- further opcodes for conditional jumps, procedure calls . . . .

[1] In the 2A machine code in Louden, for instance in page 12 or the introductory slides, the order is the opposite!

Side remark: 3A machine code

Possible format

```
OP source1 source2 dest
```

- but then: what's the *difference* to 3A *intermediate* code?
- apart from a more restricted instruction set:
- restriction on the operands, for example:
  - only one of the arguments allowed to be a memory cell
  - no fancy addressing modes (indirect, indexed ... see later) for memory cell, only for registers
- not "too much" memory-register traffic back and forth per machine instruction
- example:

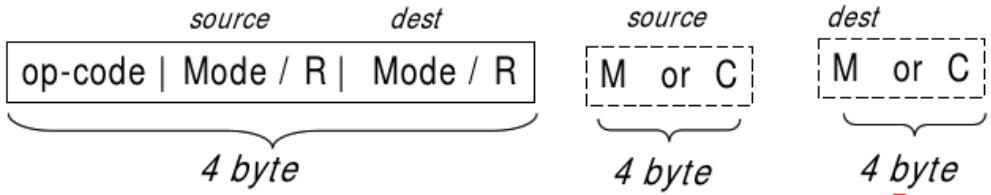
```
&x = &y + *z
```

may be 3A-intermediate code, but not 3A-machine code

"Cost model"

- "optimization": need some well-defined "measure" of the "quality" of the produced code
- interested here in execution time
- not all instructions take the same time
- estimation of execution
- cost factors:
  - *size* of the instruction
    - it's here not about code size, but
    - instructions need to be *loaded*
    - longer instructions => perhaps longer load
  - address modes (as *additional costs*: see later)
    - registers vs. main memory vs. constants
    - direct vs. indirect, or indexed access
- factor outside our control/not part of the cost model: effect of *caching*

Instruction modes and additional costs



Mode	Form	Address	Added cost	
absolute	M	M	1	
register	R	R	0	
indexed	c(R)	$c + cont(R)$	1	
indirect register	*R	$cont(R)$	0	
indirect indexed	*c(R)	$cont(c + cont(R))$	1	
literal	$\#M$	the *value* M	1	only for source

- indirect: useful for elements in "records" with known off.set
- indexed: useful for slots in arrays

Examples a := b + c

Using registers

```
MOV b, R0          // R0 = b
ADD c, R0
// R0 = c + R0
MOV R0, a          // a = R0

cost = 6
```

Memory-memory ops

```
MOV b, a           // a = b
ADD c, a           // a = c + a

cost = 6
```

Data already in registers

```
MOV *R1, *R0       // *R0 = *R1
ADD *R2, *R1
// *R1 = *R2 + *R1

cost = 2
```

Assume R0, R1, and R2 contain *addresses* for a, b, and c

Storing back to memory

```
ADD R2, R1
// R1 = R2 + R1
MOV R1, a          // a = R1

cost = 3
```

Assume R1 and R2 contain *values* for b, and c

Outline

CFG

basically: graph with

- nodes = basic blocks
- edges = (potential) jumps (and "fall-throughs")

- here (as often): CFG on 3AC (linear intermediate code)
- also possible CFG on intermediate code,
- or even:
  - CFG extracted from AST
  - here: the opposite: synthesizing a CFG from the linear code
- explicit data structure (as another intermediate representation) or implicit only.

From 3AC to CFG: "partitioning algo"

- remember: 3AC contains *labels* and (conditional) jumps

$\Rightarrow$ algo rather straightforward

- the only complication: some labels can be ignored
- we ignore procedure/method calls here
- concept "leader" representing the nodes/basic blocks

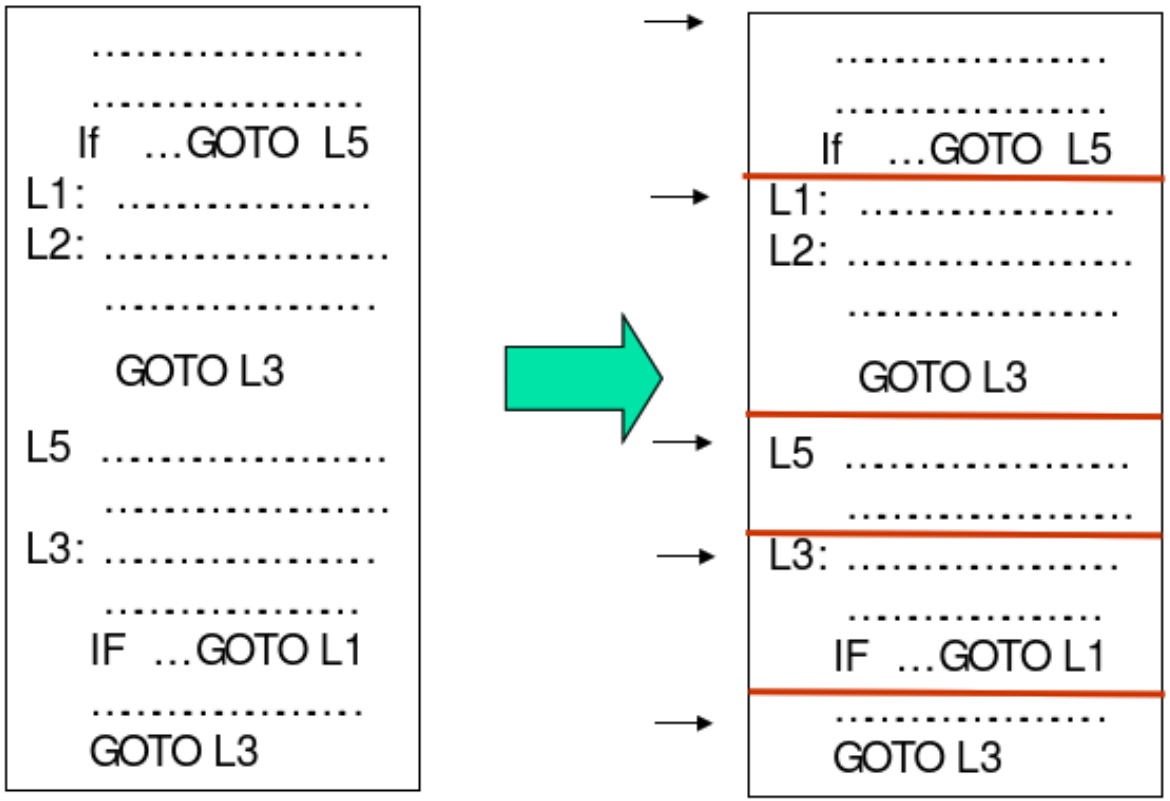
Leader

- first line is a leader
- **GOTO** *i*: line labelled *i* is a leader
- instruction *after* as **GOTO** is a leader

Basic block

instruction sequence from (and including) one leader to (but excluding) the next leader or to the end of code

Partitioning algo

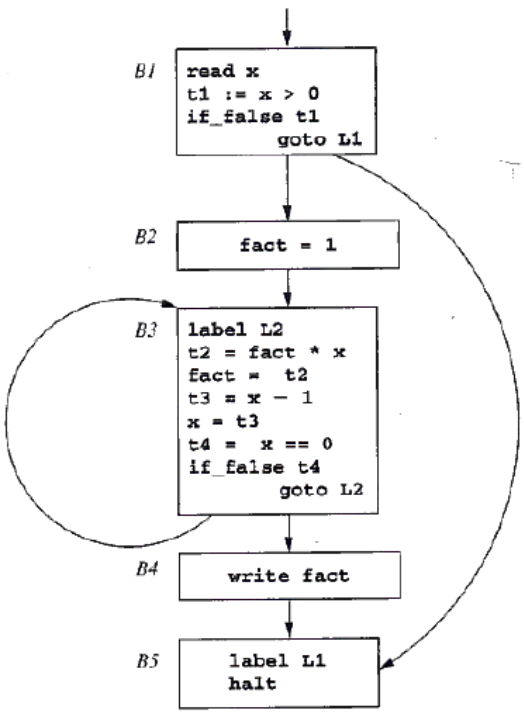


- note: no line jumps to L_2

3AC for faculty

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label = L2
t2 = fact * x
fact = t2
t3 = x -1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Faculty: CFG



- goto/conditional goto: never *inside* block
- not every block
  - ends in a goto
  - starts with a label
- ignored here: function/method calls
- *intra-procedural* control-flow graph

Levels of analysis

- here: *three* levels where to apply code analysis / optimization
  1. local: per basic block (block-level)
  2. global: per function body/intra-procedural CFG[2]
  3. inter-procedural: really global, whole-program analysis

- the "more global", the more *costly* the analysis and, especially the optimization (if done at all)

[2] the terminology "global" is not ideal, for instance process-level analysis (called global here) can be seen as procedure-local, as opposed to inter-procedural analysis which then is global.

Loops in CFGs

- *loop optimization*: "loops" are thankful places for optimizations
- important for analysis to detect loops (in the cfg)
- importance of *loop discovery*: not too important any longer in modern languages.

Loops in a CFG vs. graph cycles

- concept of loops in CFGs not identical with cycles in a graph[a]
- all loops are graph cycles but not vice versa

[a] Otherwise: cycle/loop detection not worth much discussion

- intuitively: loops are cycles orinating from source-level looping constructs ("while")
- goto's may lead to non-loop cycles in the CFG
- importants of loops: loops are "well-behaved" when considering certain optimizations/code transformations (goto's can destroy that...)

Loops in CFGs: definition

- remember: strongly connected components

Definition (Loop)

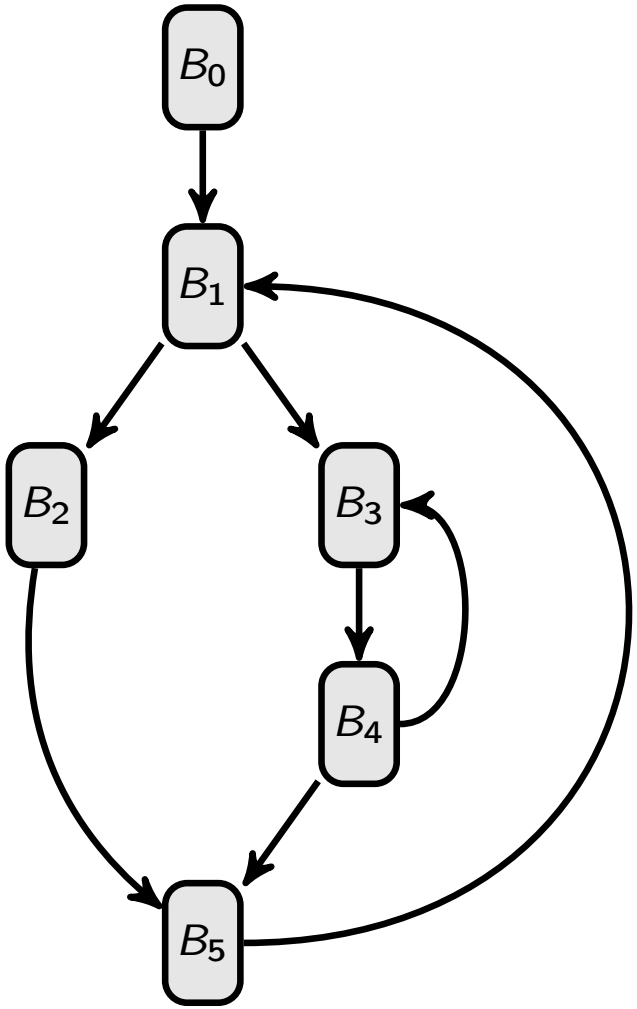
A loop L in a CFG is a collection of nodes s.t.:

- strongly connected component (with edges completely in L
- 1 (unique) *entry* node of L, i.e. no node in L has an incoming edge[a] from outside the loop *except* the *entry*

[a] alternatively: general reachability

- often also: "root" node of a CFG (there's only one) is *not* itself an entry of a loop

Loop



- Loops:
  - $\{B_3, B_4\}$
  - $\{B_4, B_3, B_1, B_5, B_2\}$
- Non-loop:
  - $\{B_1, B_2, B_5\}$
- unique entry marked red

Loops as fertile ground for optimizations

```
while (i < n) {i++; A[i] = 3*k }
```

- possible optimizations
  - move `3*k` "out" of the loop
  - put frequently used variables into *registers* while in the loop (like i)
- when moving out computation from the look:
- put "right in front of the loop"

$\Rightarrow$ add extra node/basic block in front of the *entry* of the loop[3]

[3] that's one of pragmatic motivation for unique entry.

Loop non-examples

Data flow analysis in general

- general *analysis technique* working on CFGs
- many concrete forms of analyses
- such analyses: basis for (many) *optimizations*
- *data*: info stored in memory/temporaries/registers etc.
- *control*:
  - movement of the instruction pointer
  - abstractly represented by the CFG
    - inside elementary blocks: increment of the IS
    - edges of the CFG: (conditional) jumps
    - jumps together with RTE and calling convention

Data flowing from (a) to (b)

Given the control flow (normally as CFG): is it possible (or is it guaranteed) that some "data" originating at one control-flow point (a) reach another control flow point (b).

Data flow as abstraction

- data flow analysis: fundamental and important *static* analysis technique
- it's impossible to decide statically if data from (a) *actually* "flows to" (b)

$\Rightarrow$ approximative

- therefore: work on the CFG: if there is two options/outgoing edges: *consider both*
- Data-flow answers therefore approximatively
  - if it's *possible* that the data flows from (a) to (b)
  - it's *neccessary* or unavoidable that data flows from (a) to (b)
- for *basic blocks*: exact answers possible[4]

[4] static simulation here was done for basic blocks only and for the purpose of *translation*. The translation of course needs to be exact, non-approximative. Symbolic evaluation also exist (also for other purposes) in more general forms, especially also working approximatively on abstractions.

Data flow analysis: Liveness

- prototypical / important data flow analysis
- especially important for register allocation

Basic question

When (at which control-flow point) can I be *sure* that I don't need a specific variable (temporary, register) any more?

- optimization: if sure that not needed in the future: register can be used otherwise

Definition (Live)

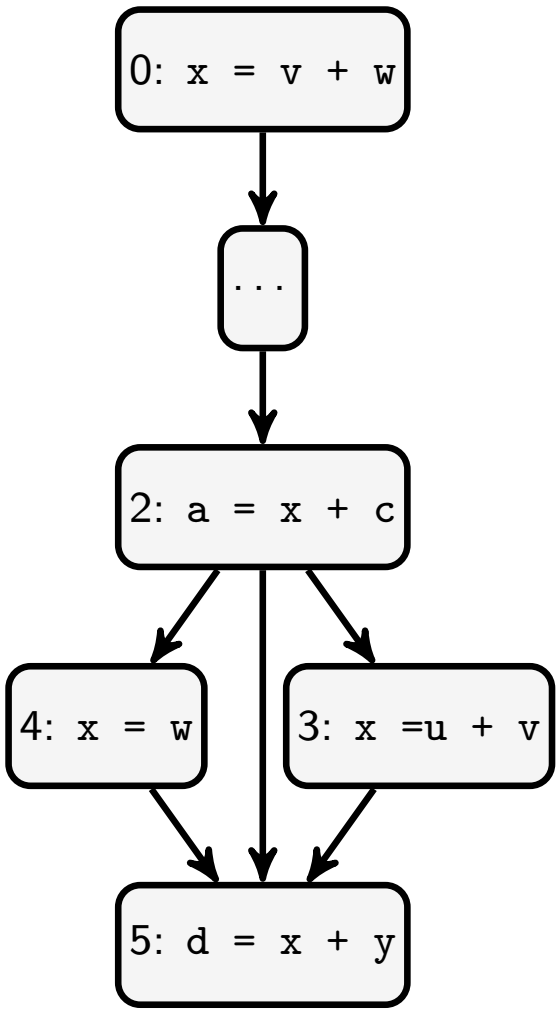
a variable is live at a given control-flow point if there *exists* an execution starting from there, where the variable is *used* in the future.[a]

[a] That corresponds to static liveness (the notion the static liveness analysis deals with. A variable in a given concrete execution of a program is *dynamically live* if in the future, it is still needed (or, for non-deterministic programs: if there exists a future, where it's still used. Dynamic liveness is undecidable, obviously.

Definitions and uses of variables

- when talking about "varaibles": also temporary variables are meant.
- basic notions underlying most data-flow analyses (including liveness analysis)
- here: def's and uses of *variables* (or temporaries etc)
- all data, including intermediate results) has to be stored somewhere, in variables, temporaries, etc.
- a "definition" of x = assignment to x (store to x)
- a "use" of x: read content of x (load x)
- variables can occur more than once, so
- a definition/use refers to *instances* or *occurrences* of variables ("use of x in line l " or "use of x in block b ")
- same for liveness: "x is live here, but not there"

Defs, uses, and liveness



- x is "defined" (= assigned to) in 0 and 4
- x is live "in" (= at the end of) block 2, as it *may* be *used* in 5
- a *non-live* variable at some point: "dead" which means: the corresponding memory can be reclaimed.
- *note*: here liveness across block-boundaries = "global" (but blocks contain only one instruction here)

Def-use or use-def analysis

- use-def: given a "use": determine all possible "definitions"[5]
- def-use: given a "def": determine all possible "uses"
- for straight-line-code/inside one basic block
  - deterministic: each line has has exactly one place where a given variable has been assigned to last (or else not assigned to in the block). Equivalently for uses.
- for whole CFG:
  - approximative ("may be used in the future")
  - more advanced techiques (caused by presence of loops/cycles)
- def-use analysis:
  - closely connected to liveness analysis (basically the same)
  - prototypical data-flow question (same for use-def analysis), related to many data-flow analyses (but not all)
- Side remark: *Static single-assignment* (SSA) format:
  - at most one assignment per variable.
  - "definition" (place of assignment) for each variable thus *clear* from its name

[5] remember: "defs" and "uses" refer to instances of definitions/assignments in the graph

Calculation of def/uses (or liveness . . .)

- three levels of complication
  1. inside basic block
  2. branching (but no loops)
  3. Loops
  4. [even more complex: inter-procedural analysis]

For SLC/inside basic block

- determnistic result
- simple "one-pass" treatment enough
- similar to "static simulation"

For whole CFG

- iterative algo needed
- dealing with non-determinism: over-approximation
- "closure" algorithms, similar to the way e.g., dealing with *first* and *follow* sets
- = fix-point algorithms

Inside one block: optimizing use of temporaries

- simple setting: *intra*-block analysis & optimization, only
- temporaries:
  - symbolic representations to hold intermediate results
  - generated on request, assuming unbounded numbers
  - intentions: use registers
- limited about of register available (platform dependent)

Assumption

- temp's *don't transfer* data accross blocks ($\neq$ program var's)

$\Rightarrow$ temp's *dead* at the beginning and at the end of a block

- but: variables have to be *considered live* at the end of a block (block-local analysis, only)

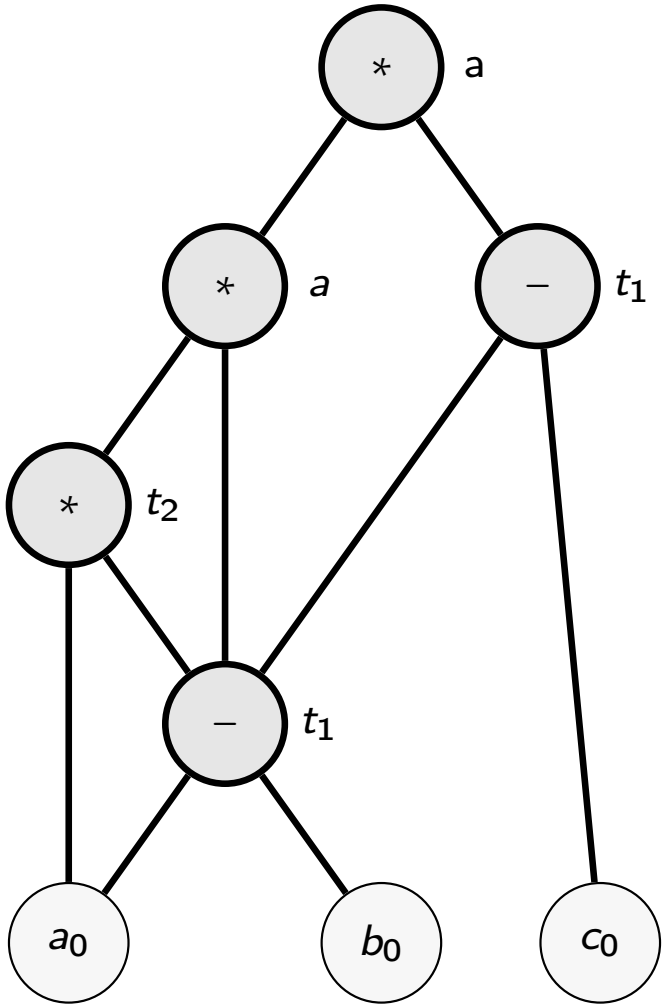
Intra-block liveness

```
t1 := a - b
t2 := t1 * a
a  := t1 * t2
t1 := t1 - c
a  := t1 * a
```

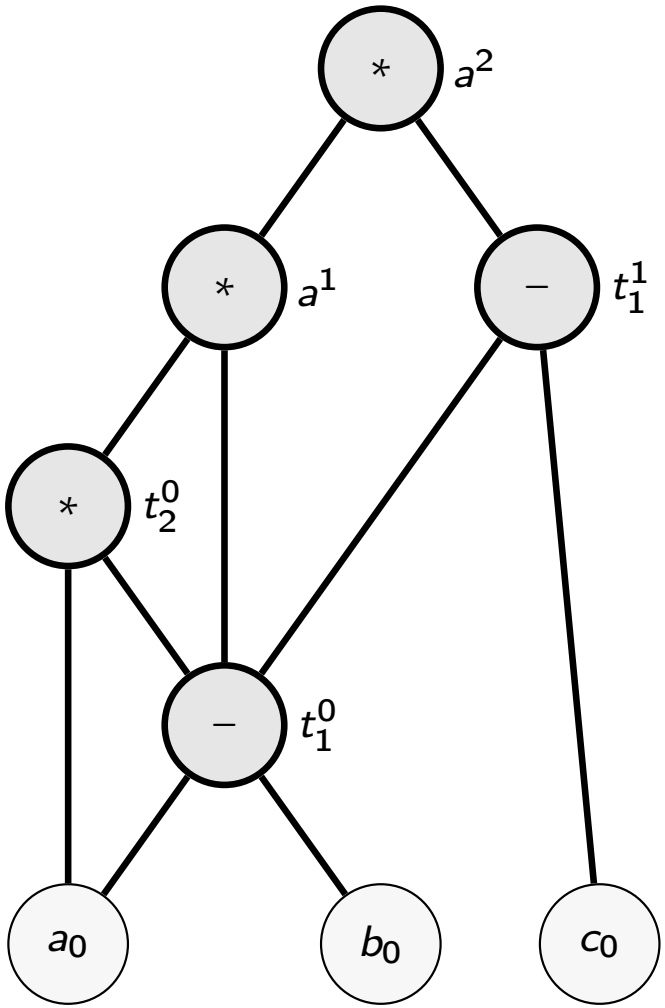
- neither temp's nor vars: "single assignment",
- but first occurrence of a temp in a block: a definition (but for temps it would often be the case)
- let's call operand: variables or temp's
- *next use* of an operand:
- uses of operands: on the lhs's, definitions on the rhs's
- not good enough to say "t_1 is live in line 4"

Note: the TAIC may allow also literal constants as operator arguments, they don't play a role right now.

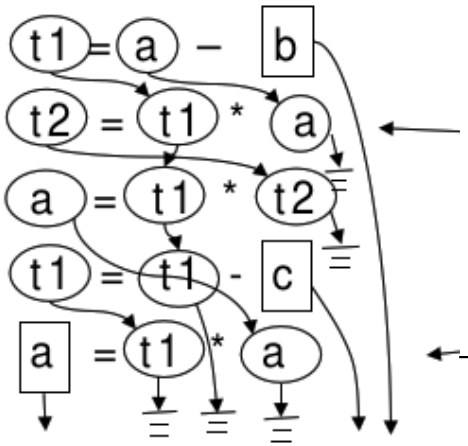
DAG of the block



- no linear order (as in code), only partial order
- *the* next use: meaningless
- but: *all "next" uses* visible (if any)
- node = occurrences of a variable on the rhs ("definitions")
- e.g.: the "lower node" for "defining" *assigning to t_1 has /three* uses


*
a^2
*
a^1
−
t_1^1
*
t_2^0
−
t_1^0
a_0
b_0
c_0

Intra-block liveness: idea of algo



- liveness-status of an operand: *different* from lhs vs. rhs in a given instruction[a]
- informal definition: an operand is live at some occurrence, if it's used some place in the future

[a] but if one occurrence of (say) x in a rhs $x + x$ is live, so is the o occurrence.

Definition (consider statement $x_1 := x_2\ op\ x_3$)

- A variable x is live at the *beginning* of $x_1 := x_2\ op\ x_3$, if
  1. if x is the same variable than x_2 or x_3, or
  2. if x live at its *end*, provided x and x_1 are different variables
- A variable x is live at the *end* of an instruction,
  - if it's live at *beginning of the next* instruction
  - if no next instruction
    - temp's are dead
    - user-leval variables are (assumed) live

Previous "inductive" definition

expresses liveness status of variables *before* a statement dependent on the liveness status of variables *after* a statement (and the variables used in the statement)

- *core* of a straightforward iterative algo
- simple backward scan[6]
- the algo we sketch:
  - not just boolean info (live = yes/no), instead:
  - operand live?
    - yes, and with next use inside is block (and indicate instruction where)
    - yes, but with no use inside this block
    - not live:
  - even more info: not just that but indicate, where's the next use

[6] Remember: intra-block/SLC. In the presence of loops/analysing a complete CFG, a simple 1-pass does not suffice. More advanced techniques ("multiple-scans" = fixpoint calculations) are needed then.

Algo: dead or alive (binary info only)

```
// ----- initialise T  ----------------------------
  for all entries: T[i,x] := D
  except: for all variables a // but not temps
             T[n,a] := L,
//-------- backward pass ----------------------------
for instruction i =  n-1 down to 0
   let current instruction i: x:=y op z;
      T[i.y] := L
      T[i.z] := L
      T[i.x] := D // note order; x can ''equal'' y or z
end
```

- Data structure T: table, mapping for each line/instruction i and variable: boolean status of "live"/"dead"
- represents liveness status per variable *at the end (i.e. rhs)* of that line
- basic block: n instructions, from 1 until n, where "line 0" represents the sentry imaginary line "before" the first line (no instruction in line 0)
- *backward scan* through instructions/lines from n to 0

Algo′: dead or else: alive with next use

- More refined information
- not just binary "dead-or-alive" but next-use info

⇒ three kinds of information

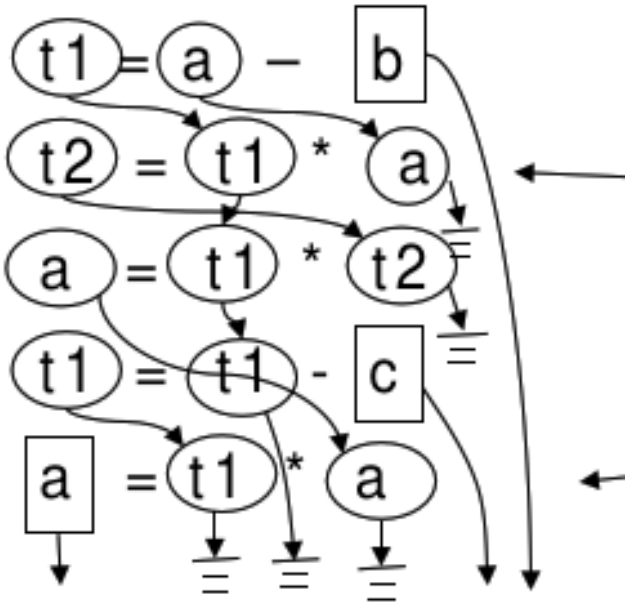
1. Dead: D
2. Live:
   - with *local* line number of *next use*: $L(n)$
   - *potential* use of outside local basic block $L(\perp)$

- otherwise: basically the same algo

```
// ----- initialise T  ---------------------------
  for all entries: T[i,x] := D
  except: for all variables a // but not temps
             T[n,a] := L(⊥),
//------- backward pass ---------------------------
for instruction i = n-1 down to 0
   let current instruction i: x := y op z;
      T[i,y] := L(i+1)
      T[i,z] := L(i+1)
      T[i,x] := D // note order; x can ''equal'' y or z
end
```

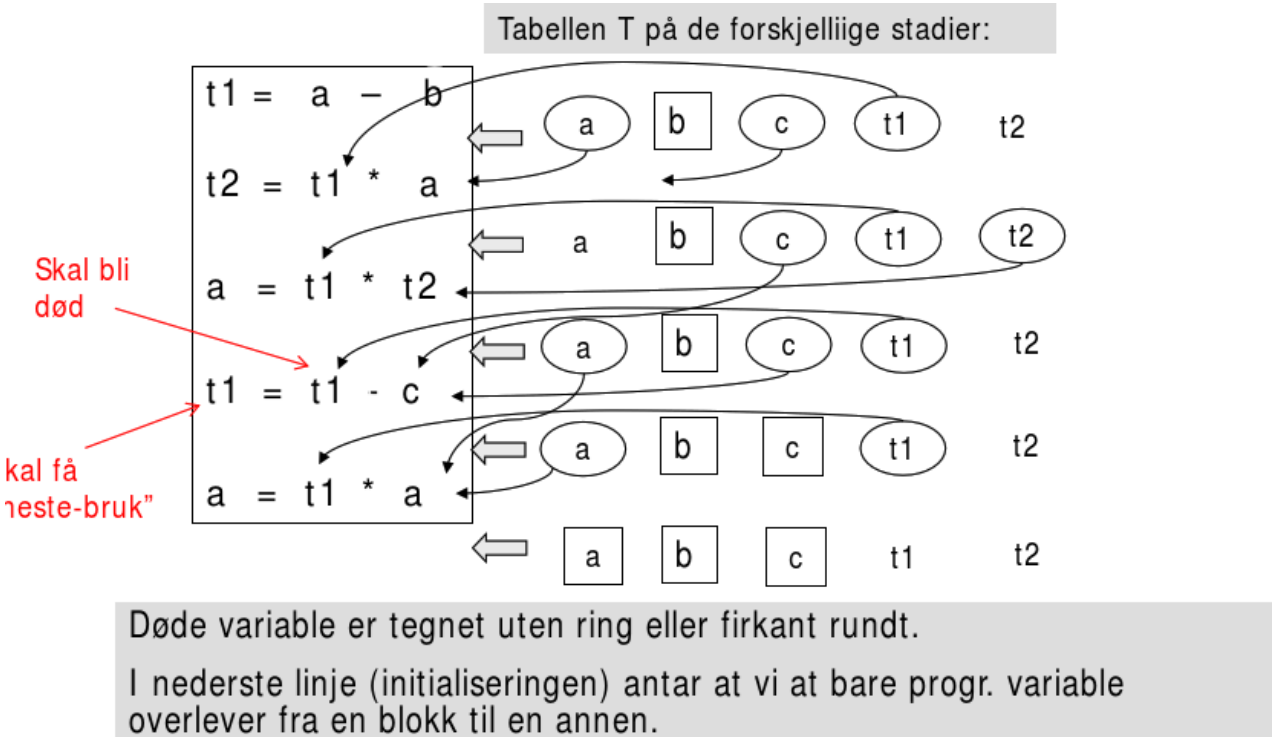
Run of the algo′

line	a	b	c	t_1	t_2
[0]	$L(1)$	$L(1)$	$L(4)$	$L(2)$	D
1	$L(2)$	$L(\perp)$	$L(4)$	$L(2)$	D
2	D	$L(\perp)$	$L(4)$	$L(3)$	$L(3)$
3	$L(5)$	$L(\perp)$	$L(4)$	$L(4)$	D
4	$L(5)$	$L(\perp)$	$L(\perp)$	$L(5)$	D
5	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D




```
t1 := a - b
t2 := t1 * a
a  := t1 * t2
t1 := t1 - c
a  := t1 * a
```

Liveness algo remarks



- here: T data structure traces (L/D) status per variable $\times$ "line"
- in the *remarks* in the notat:
  - alternatively: store liveness-status *per variable* only
  - works as well for one-pass analyses (but only without loops)
- this version here: corresponds better to global analysis: 1 line can be seen as one small basic block

Outline

Simple code generation algo

- simple algo: *intra-block* code generation
- core problem: register use
- register allocation & assignment [7]
- hold calculated values in registers longest possibe
- intra-block only ⇒ at exit:
  - all *variables* stored back to main memory
  - all temps assumed "lost"
- remember: assumptions in the intra-block liveness analysis

[7] some distinguish register allocation: "should the data be held in register (and how long)" vs. register assignment: "which of available register to use for that"

Limitations of the code generation

- local intra block:
  - no analysis across blocks
  - no procedure calls etc
- no complex data structures
  - arrays
  - pointers
  - ...

some limitations on how the algo itself works for one block

- read-only variables: never put in registers, even if variable is *repeatedly* read
  - algo works only with the temps/variables given and does not come up with new ones
  - for instance: DAGs could help
- no *semantics* considered
  - like *commutativity*: $a + b$ equals $b + a$

Purpose and "signature" of the *getreg* function

- one *core* of the code generation algo
- simple code-generation here ⇒ simple *getreg*

getreg function

available: *liveness/next-use* info

Input: TAIC-instruction $x := y$ **op** z

Output: return *location* where x is to be stored

- location: register (if possible) or memory location

Coge generation invariant

it should go without saying . . . :

Basic safetey invariant

At each point, "live" variables (with or without next use in the current block) must exist in at least one location

- another (specific) invariant: the location returned by getreg: the one where the rhs of a 3AIC assignment ends up

Register and address-descriptors

- code generation/*getreg*: keep track of
  1. register contents
  2. addresses for names

Register descriptor

- tracking current content of reg's (if any)
- consulted when new reg needed
- as said: at block entry, assume all regs unused

Address descriptor

- tracking location(s) where current value of name can be found
- possible locations: register, stack location, main memory
- >1 location possible (but not due to overapproximation, exact tracking)

Code generation algo for $x := y$ **op** z

1. determine location (preferably register) for result

```
L = getreg( ``x := y op z'')
```

2. make sure, that the value of y is in L :
   - consult address descriptor for $y \Rightarrow$ current locations y' for y
   - choose the best location y' from those (register)
   - if value of y *not* in L, generate

```
MOV y', L
```

3. generate

```
OP z', L  // z': a current location of z (prefer reg's)
```

   - update address descriptor $x \mapsto L$
   - if L is a reg: update reg descriptor $L \mapsto x$

4. exploit liveness/next use info: update register descriptors

Skeleton code generation algo for $x := y$ **op** z

```
l = getreg(``x:= y op z'') // target location for x
if l ∉ Ta(y) then let ly ∈ Ta(y)) in emit ("MOV ly, l");
let z′ ∈ Ta(z)) in emit ("OP z′, l");
```

- "skeleton"
  - non-deterministic: we igored how to choose z' and y'
  - we ignore *book-keeping* in the *name* and *address* descriptor tables
  - details of *getreg* hidden.

Non-deterministic code generation algo for $x := y$ **op** z

```
L = getreg(``x:= y op z'') // generate target location for x
if L ∉ T_a(y)
then let y' ∈ T_a(y)) // pick a location for y
       in emit (MOV y', L)
else skip;
let z' ∈ T_a(z)) in emit (``OP z', L'');
T_a := T_a[x ↦ L];
if   L is a register
then T_r := T_r[L ↦ x]
```

Code generation algo for $x := y$ **op** z

```
l = getreg("i: x := y op z")   // i for instructions line number/la
if l ∉ T_a(y)
then let l_y = best (T_a(y))
       in  emit ("MOV l_y, l")
else skip;
let l_z = best (T_a(z))
in emit ("OP l_z, l");
T_a := T_a\(_ ↦ l);
T_a := T_a[x ↦ l];
T_r := T_r[l ↦ x];

if  ¬T_live[i, y] and T_a(y) = r then T_r := T_r\(r ↦ y)
if  ¬T_live[i, z] and T_a(z) = r then T_r := T_r\(r ↦ z)
```

Code generation algo for $x := y$ **op** z (Notat)

```
l = getreg(" x := y op z")
if l ∉ T_a(y)
then let l_y = best (T_a(y))
       in  emit ("MOV l_y, l")
else skip;
let l_z = best (T_a(z))
in emit ("OP l_z, l");
   T_a := T_a\(_ ↦ l);
   T_a := T_a[x ↦ l];
   T_r := T_r[l ↦ x]
```

Exploit liveness/next use info: recycling registers

- register descriptors: don't update themselves during code generation
- once set (for instance as $R_0 \mapsto t$), the info stays, unless reset
- thus in step 4 for $z := x$ **op** y:

to exploit liveness info by recycling reg's

if y and/or z are currently

- *not live* and are
- in *registers*,

$\Rightarrow$ "wipe" the corresponding info from the corresponding register descriptors

- side remark: for address descriptor
  - no such "wipe" neeed, because it won't make a difference (y and/or z are not-live anyhow)
  - their address descriptor wont' be consulted further in the block

getreg algo: $x := y$ **op** z

- goal: return a location for x
- basically: check possibilities of register uses,
- start with the cheapest option

do the following steps, in that order

1. in place if x is in a register already (and if that's fine otherwise), then return the register.
2. new register if there's an unsused register: return that
3. purge filled register choose more or less cleverly a filled register and save its content, if needed, and return that register
4. use main memory if all else fails

getreg algo: $x := y$ **op** z in more details

1. if
   - y in register R
   - R holds *no alternative names*
   - y is *not live* and has no next use after the 3AIC instruction
   - $\Rightarrow$ return R
2. else: if there is an empty register R': return R'
3. else: if
   - x has a next use [or operator requires a register] $\Rightarrow$
     - find an occupied register R
     - store R into M if needed (MOV R, M))
     - don't forget to update M 's address descriptor, if needed
     - return R
4. else: x not used in the block *or* no suituable occupied register can be found
   - return x as location L

- choice of purged register: heuristics
- remember (for step 3): registers may contain value for > 1 variable $\Rightarrow$ multiple MOV's

Sample TAIC

```
d := (a-b) + (a-c) + (a-c)
```

```
t := a - b
u := a - c
v := t + u
d := v + u
```

line	a	b	c	d	t	u	v
[0]	$L(1)$	$L(1)$	$L(2)$	D	D	D	D
1	$L(2)$	$L(\bot)$	$L(2)$	D	$L(3)$	D	D
2	$L(\bot)$	$L(\bot)$	$L(\bot)$	D	$L(3)$	$L(3)$	D
3	$L(\bot)$	$L(\bot)$	$L(\bot)$	D	D	$L(4)$	$L(4)$
4	$L(\bot)$	$L(\bot)$	$L(\bot)$	$L(\bot)$	D	D	D

Code sequence

	3AIC	2AC	reg. descr.		addr. descriptor						
			R_0	R_1	a	b	c	d	t	u	v
[0]			$\perp$	$\perp$	a	b	c	d	t	u	v
1	t := a − b	**MOV** a, R0	$[a]$		$[R_0]$						
		SUB b, R0	t		~~R_0~~				R_0		
2	u := a − c	**MOV** a, R1	·	$[a]$	$[R_0]$						
		SUB c, R1		u	~~R_0~~					R_1	
3	v := t + u	**ADD** R1, R0	v	·					~~R_0~~		R_0
4	d := v + u	**ADD** R1, R0	d					R_0			~~R_0~~
		MOV R0, d									
			R_i: unused		all var's in "home position"						

- address descriptors: "home position" not explictly needed.
- for instance: variable a always to be found "at a ", as indicated in line "0".
- in the table: only *changes* (from top to bottom) indicated
- after line 3:
  - t dead
  - t resides in R_0 (and nothing else in R_0)

  → reuse R_0

Outline

From "local" to "global" data flow analysis

- data stored in variables, and "flows from definitions to uses"
- liveness analysis
  - one prototypical (and important) data flow analysis
  - so far: *intra-block* = straight-line code
- related to
  - *def-use* analysis: given a "definition" of a variable at some place, where it is (potentially) used
  - *use-def*: (the inverse question, "reaching definitions"
- other similar questions:
  - has a value of an expression been calculated before ("available expressions")
  - will an expression be used in all possible branches ("very busy expressions")

Global data flow analysis

- block-local
  - block-local analysis (here liveness): *exact* information possible
  - block-local liveness: *1 backward scan*
  - important use of liveness: *register allocation*, temporaries typically don't survive blocks anyway
- global: working on the complete CFG

2 complications

- branching: *non-determinism*, unclear which branch is taken
- loops in the program (loops/cycles in the graph): a simple *one pass* through the graph does not cut it any longer

- *exact* answers no longer possible (undecidable)

$\Rightarrow$ work with safe approximations

- this is: general characteristic of DFA

Generalizing block-local liveness analysis

- assumptions for block-local analysis
  - all program variables (assumed) *live* at the end of each basic block
  - all temps are assumed *dead* there.[8]
- now: we do better, info accross blocks

at the end of each block:

which variables may be use used in subsequent block(s).

- now: re-use of temporaries (and thus corresponding registers) across blocks possible
- remember local liveness algo: determined liveness status per var/temp *at the end* of each "line/instruction"[9]

[8] While assuming variables live, even if they are not, is safe, the opposite may be unsafe. The code generator therefore must not reuse temporaries across blocks when doing this assumption.

[9] For sake of making a parallel one could: consider each line as individual block.

Connecting blocks in the CFG: *inLive* and *outLive*

- CFG:
  - pretty conventional graph (nodes and edges, often designated start and end node)[10]
  - *nodes* = basic blocks = contain straight-line code (here 3AIC)
  - being conventional graphs:
    - conventional representations possible
    - E.g. nodes with lists/sets/collections of immediate *successor nodes* plus immediate *predecessor nodes*
- remember: local liveness status
  - can be different *before* and *after* one single instruction
  - liveness status *before* expressed as dependent on status *after*

  $\Rightarrow$ backward scan
- Now per block: *inLive* and *outLive*

[10] For some analyses resp. algos: assumed that the only cycles in the graph are *loops*. However, the techniques presented here work generally.

- tracing / approximating set of live variables[11] at the *beginning* and *end* per basic block
- *inLive* of a block: depends on
  - *outLive* of that block and
  - the SLC inside that block
- *outLive* of a block: depends on *inLive* of the *successor* blocks

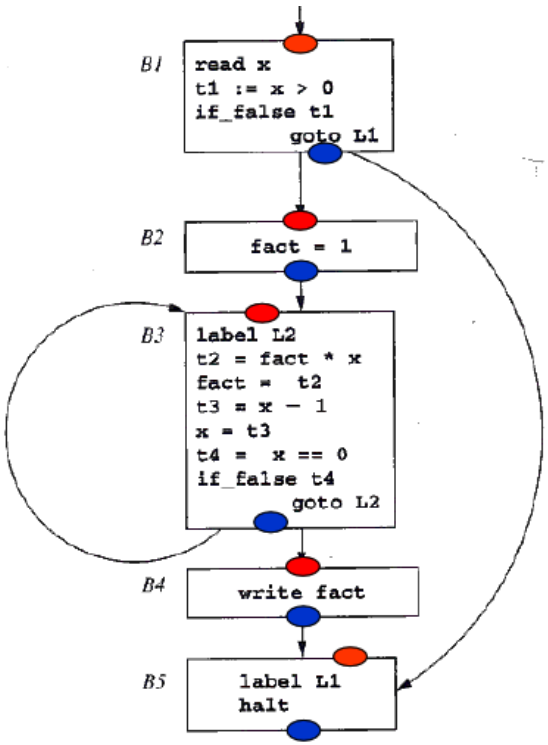
Approximation: To err on the safe side

Judging a variable (statically) live: always *safe*. Judging wrongly a variable *dead* (which actually will be used): unsafe

- goal: smallest (but safe) possible sets for *outLive* (and *inLive*)

[11] to stress "approximation": *inLive* and *outLive* contain sets of *statically* live variables. If those are dynamically live or not is undecidable.

Example: Faculty CFG



- *inLive* and *outLive*
- picture shows arrows as *successor nodes*
- needed *predecessor nodes* (reverse arrows)

node/block	predecessors
B_1	$\varnothing$
B_2	$\{B_1\}$
B_3	$\{B_2, B_3\}$
B_4	$\{B_3\}$
B_5	$\{B_1, B_4\}$

Block local info for global liveness/data flow analysis

- 1 CFG per procedure/function/method
- as for SLG: algo works backwards
- for each block: underlying block-local liveness analyis

3-values *block local* status per variable

result of block-local live variable analysis

1. *locally live* on entry: variable used (before overwritten or not)
2. *locally dead* on entry: variable overwritten (before used or not)
3. status not locally determined: variable neither assigned to nor read locally

- for efficiency: *precompute* this info, before starting the global iteration $\Rightarrow$ avoid *recomputation* for blocks in loops
- in the smallish examples: we often do it simply on-the-fly (by "looking at" the blocks' SLC)

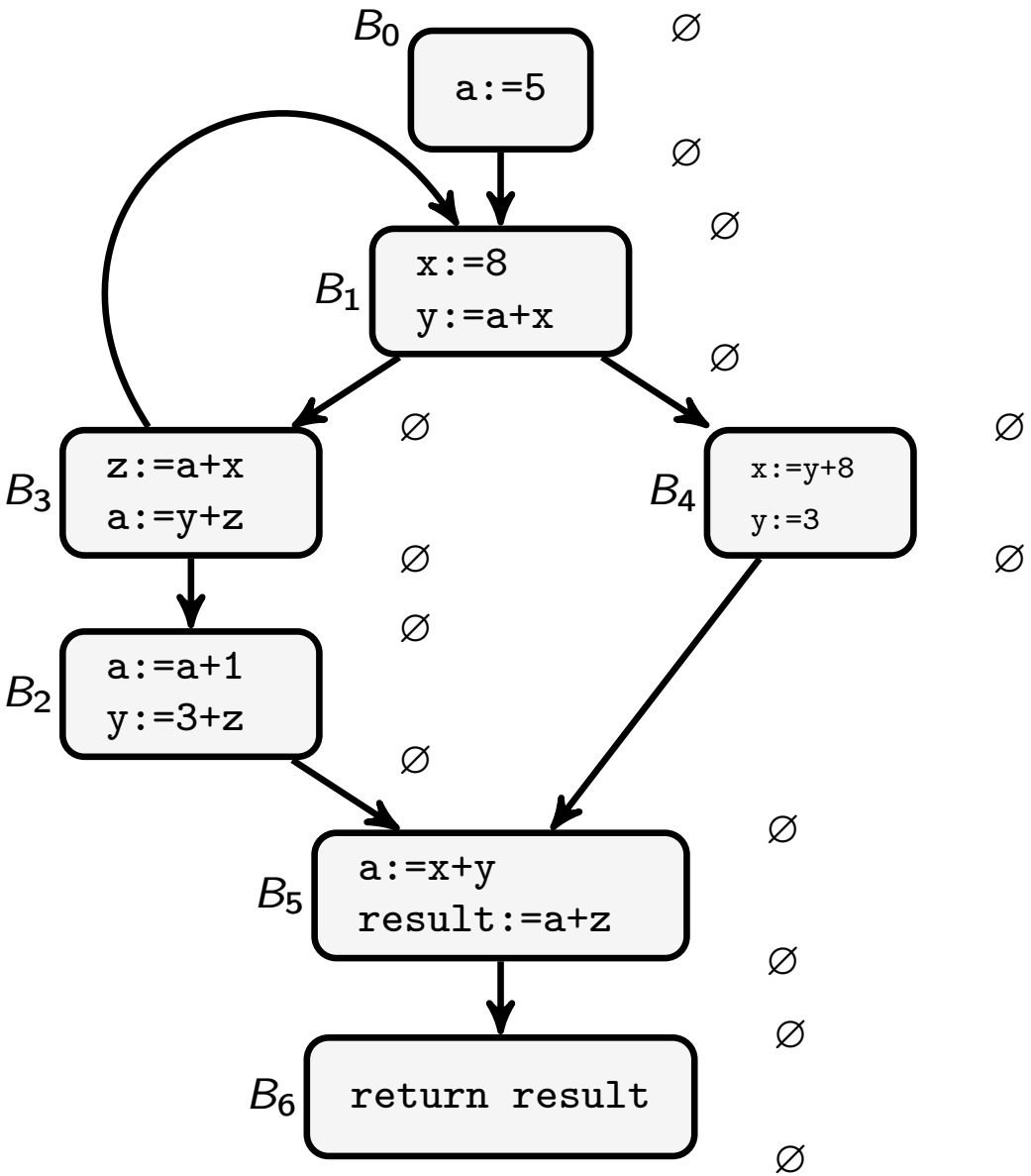
Global DFA as iterative "completion algorithm"

- different names for general approach
  - *closure* algorithm
  - *fixpoint* iteration
- basically: a big loop with
  - iterating a step approaching an intended solution by making current approximation of the solution *larger*
  - until the solution stabilizes
- similar (for example): calculation of first- and follow-sets
- often: realized as *worklist algo*
  - named after central data-structure containing the "work-still-to-be-done"
  - here possible: worklist containing nodes untreated wrt. liveness analysis (or DFA in general)

Example

```
     a := 5
L1:  x := 8
     y := a + x
     if_true x=0 goto L4
     z := a + x          // B3
     a := y + z
     if_false a=0  goto L1
     a := a + 1          // B2
     y := 3 + x
L5   a := x + y
     result := a + z
     return result       // B6
L4:  a := y + 8
     y := 3
     goto L5
```

CFG: initialization



- *inLive* and *outLive*: *initialized* to ∅ everywere
- note: start with (most) *unsafe* estimation
- extra (return) node
- but: analysis here *local per procedure*, only

General schema

Initialization start with the "minimal" estimation ($\varnothing$ everywhere)

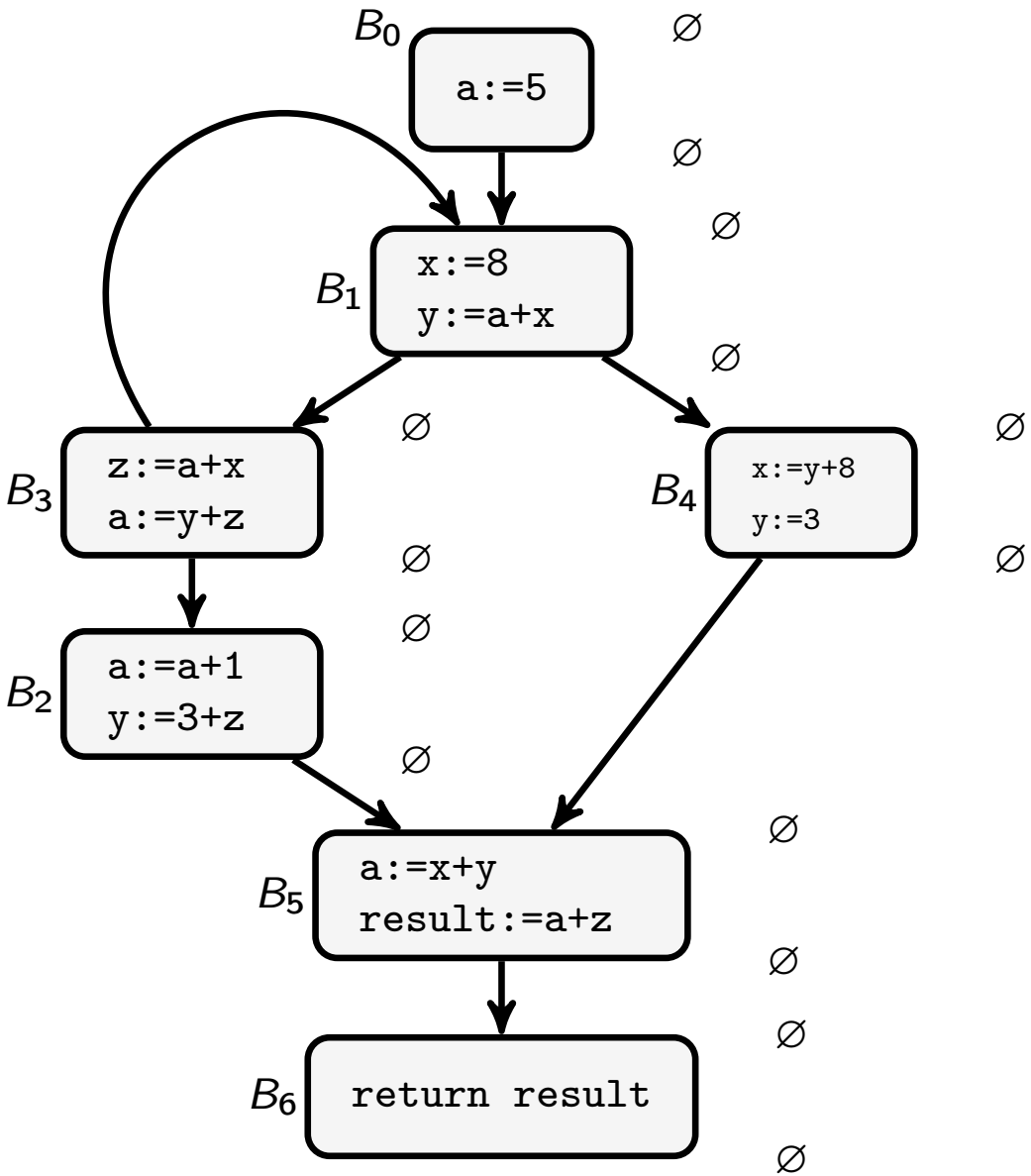
Loop pick one node & update (= enlarge) liveness estimation in connection with that node

Until finish upon stabilization. no further enlargement

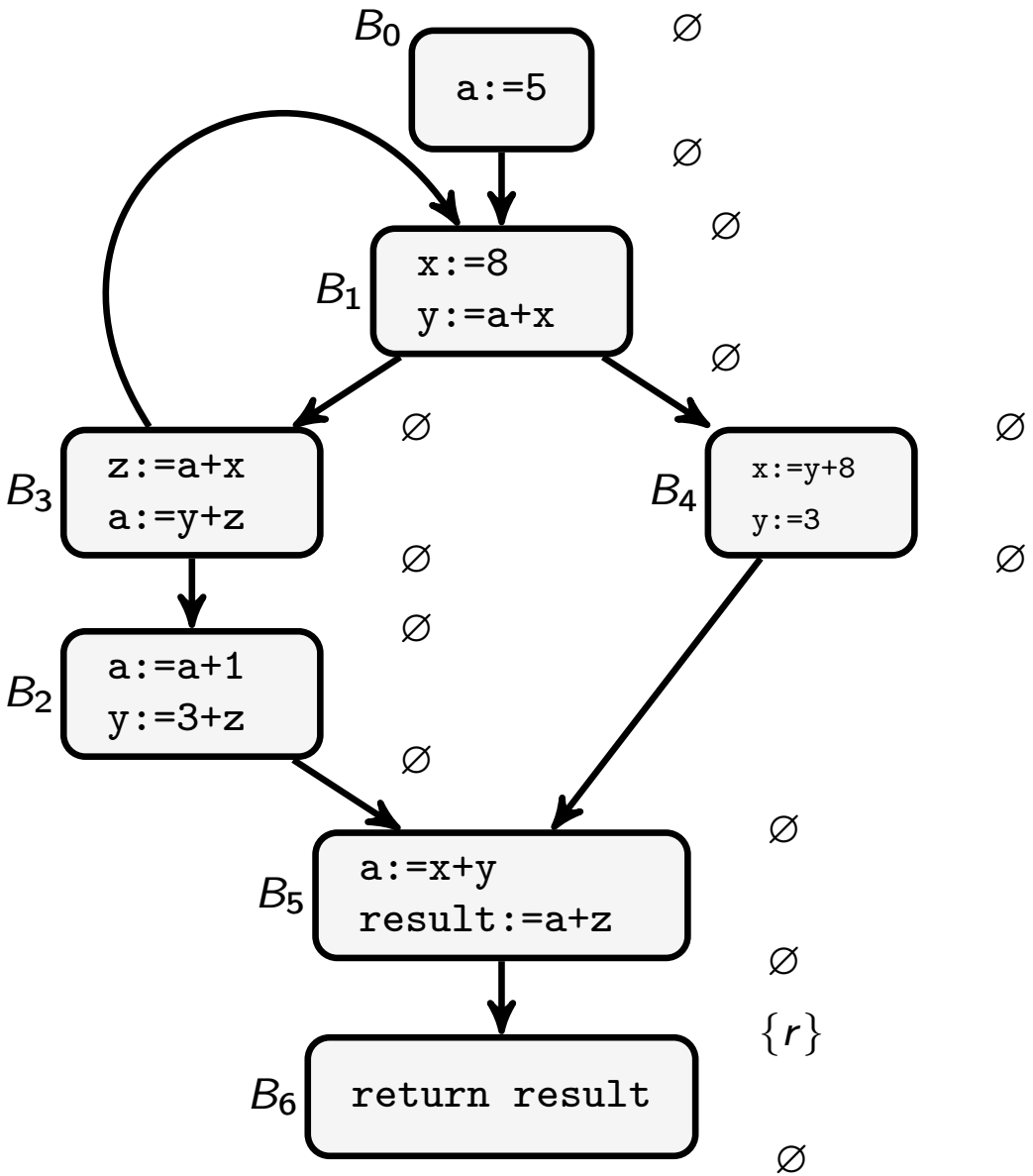
- order of treatment of nodes: in princple arbitrary[12]
- in tendency: following edges backwards
- comparison: for linear graphs (like inside a block):
  - no repeat-until-stabilize loop needed
  - 1 simple backward scan enough

[12] There may be more efficient and less efficient orders of treatment.

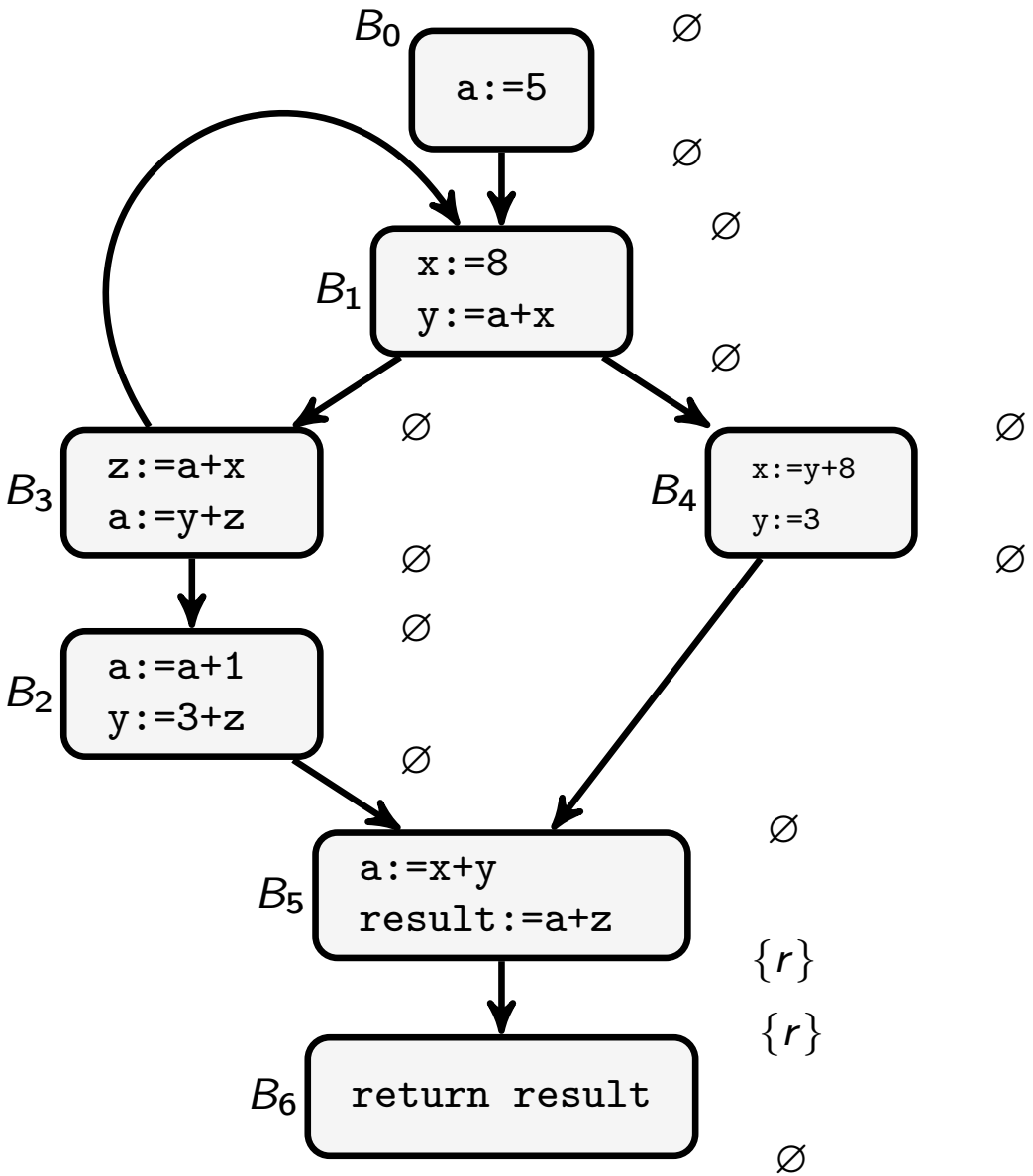
Liveness: run



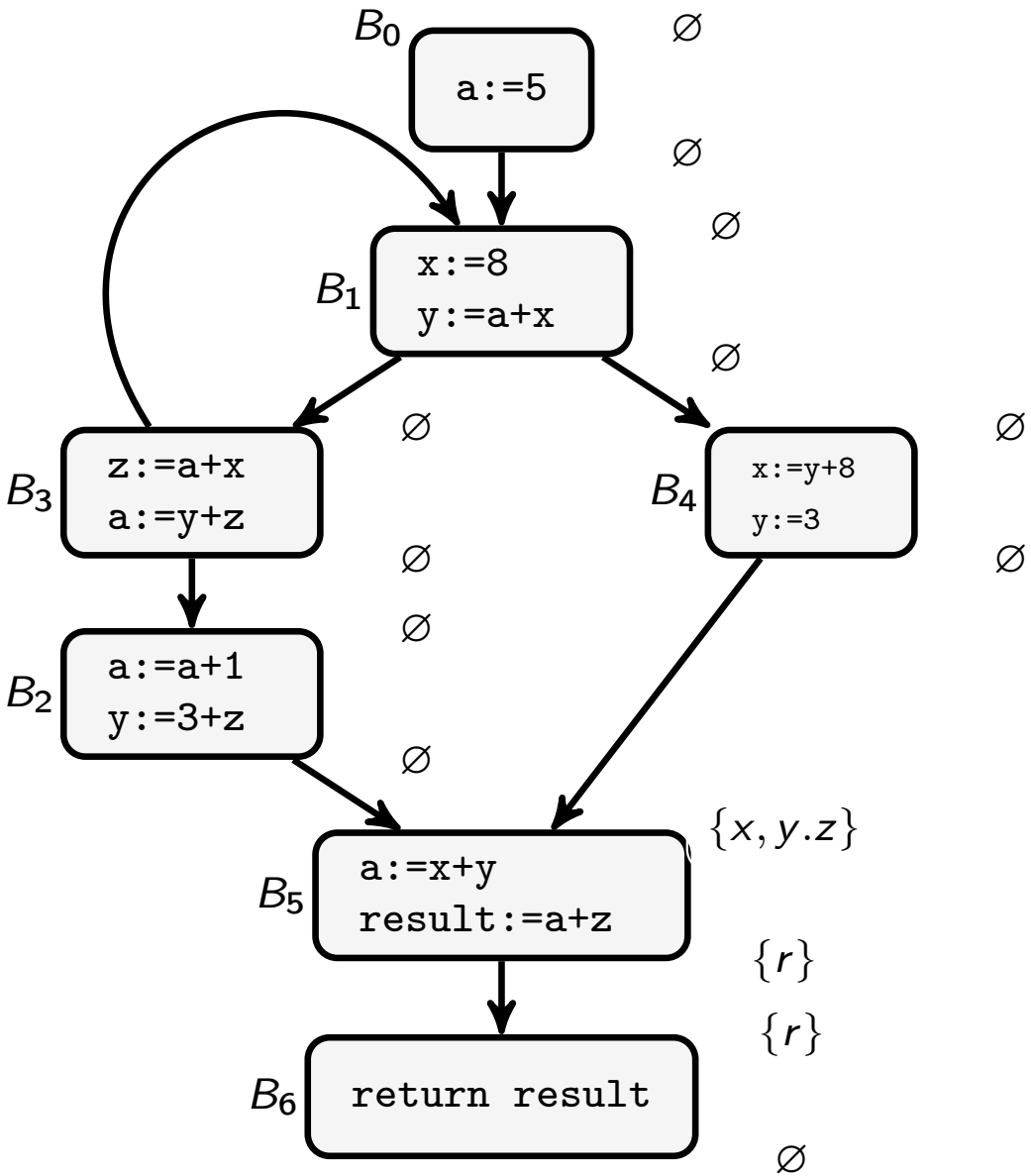
Liveness: run



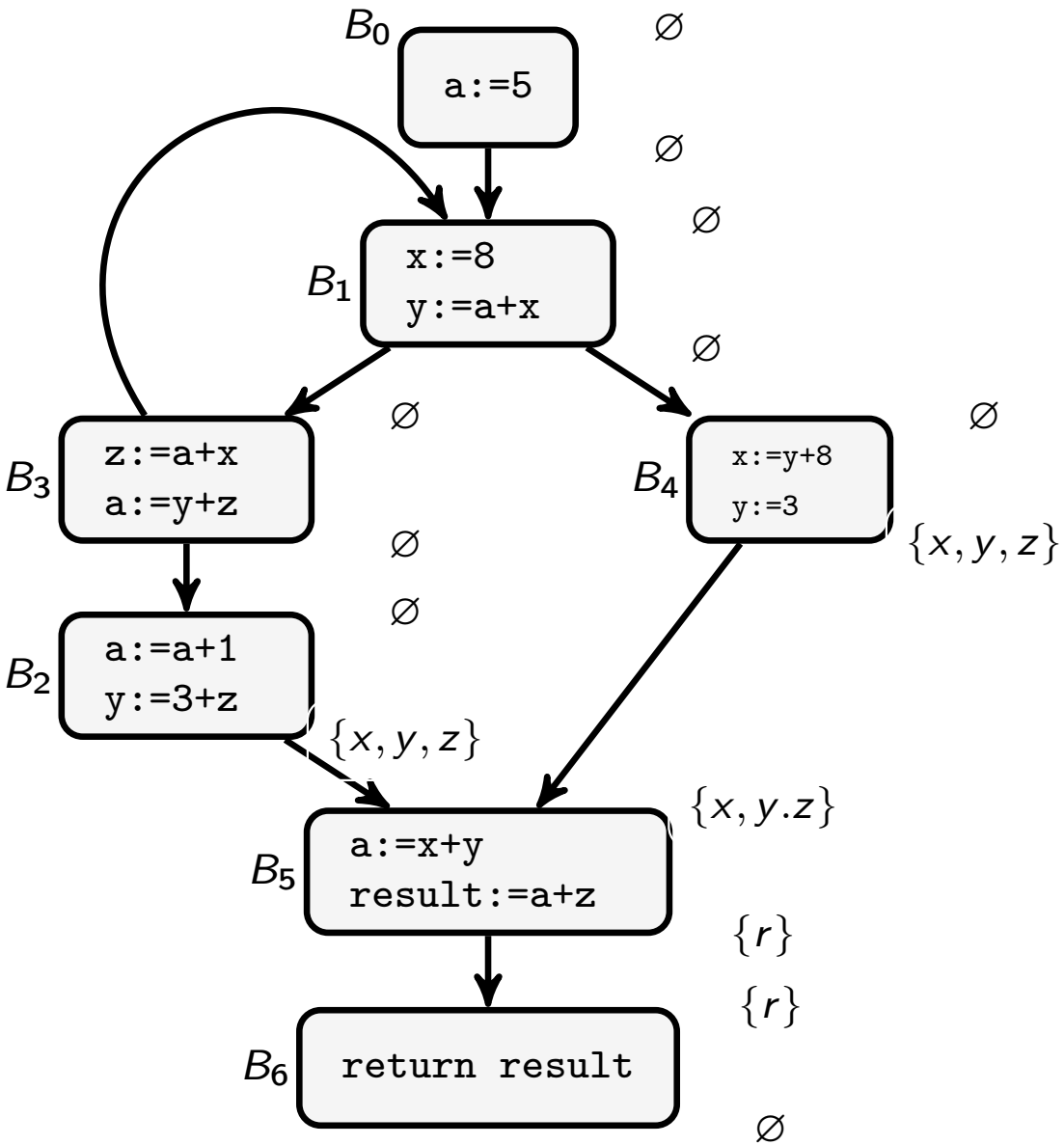
Liveness: run



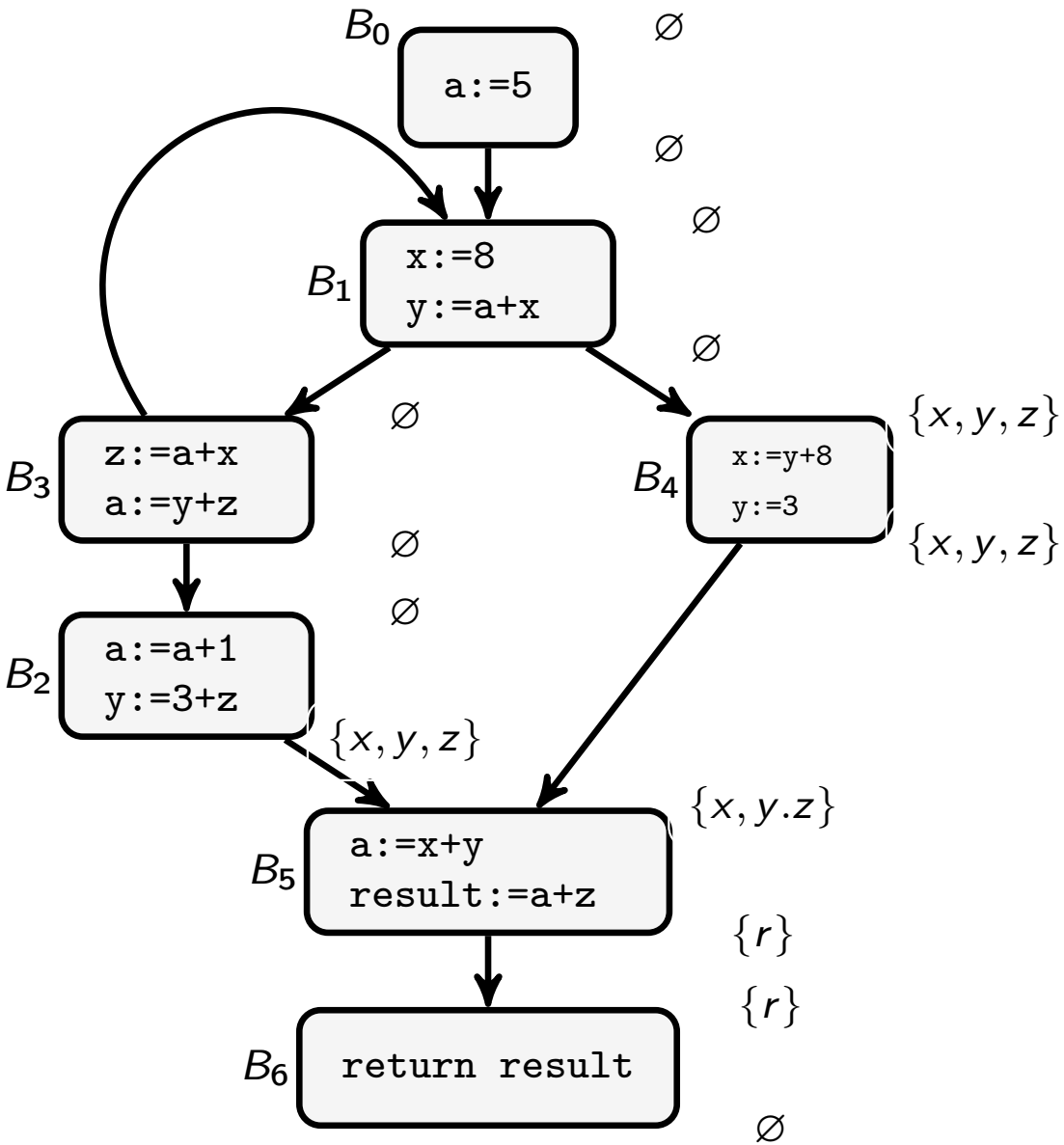
Liveness: run



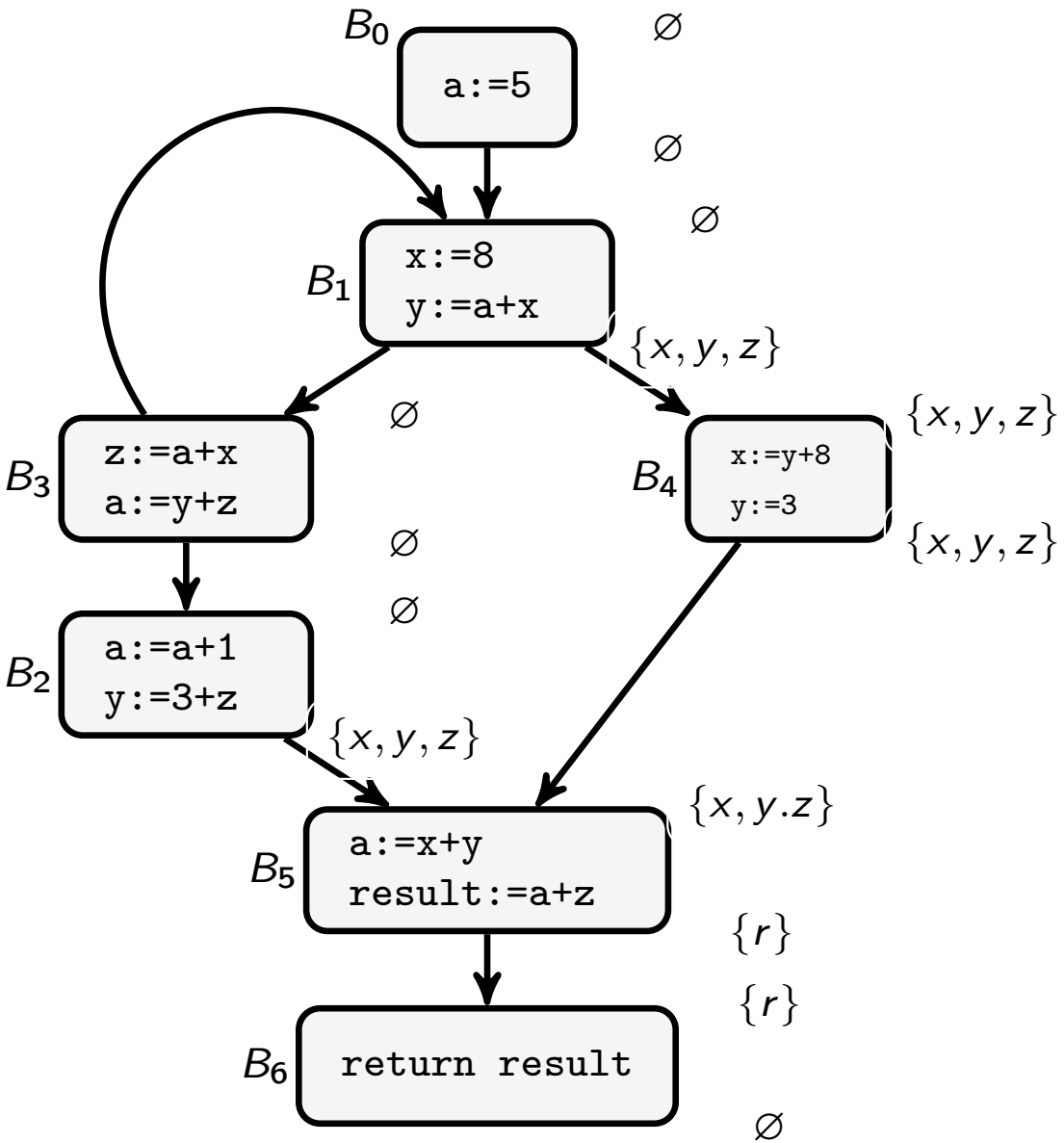
Liveness: run



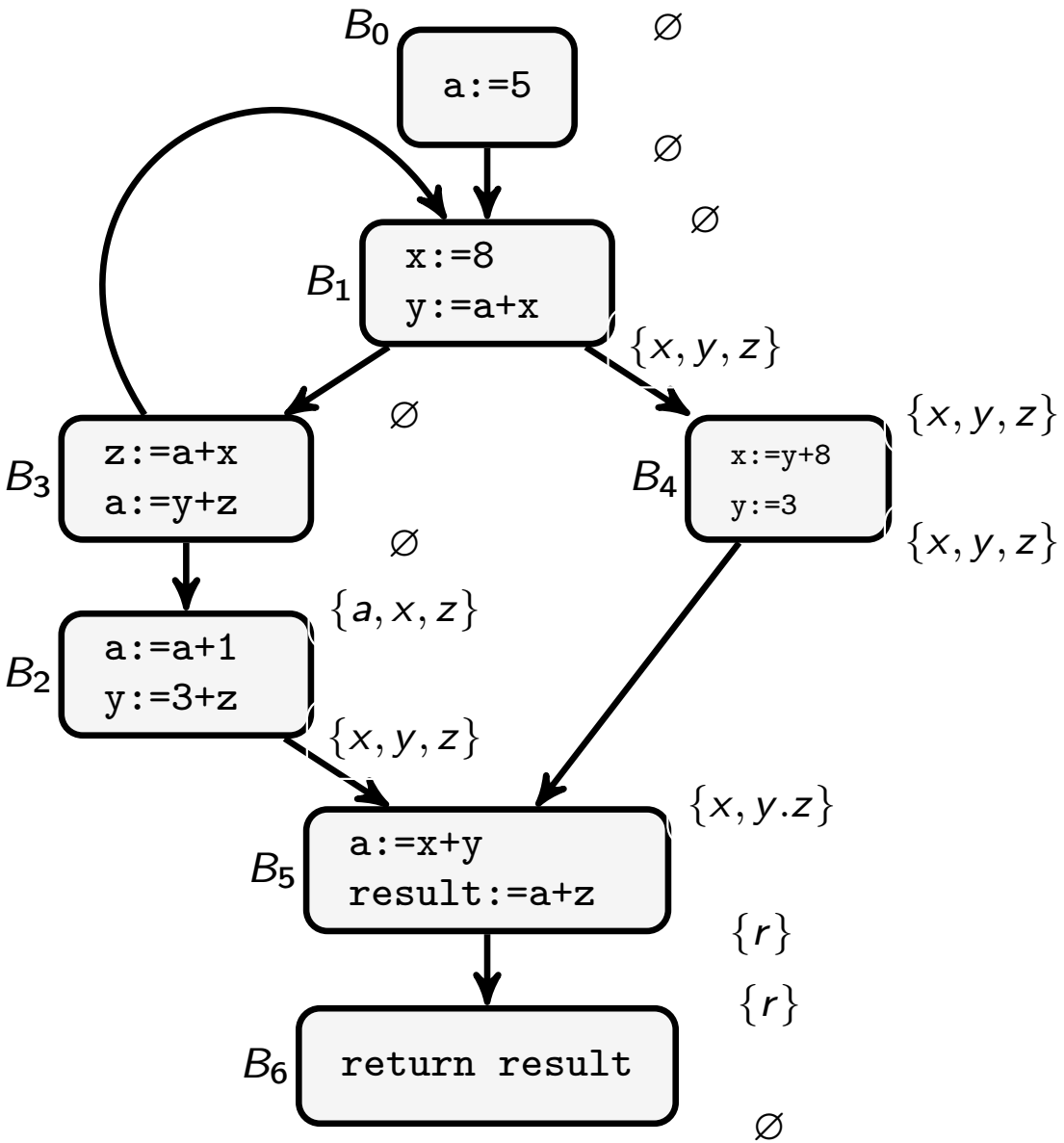
Liveness: run



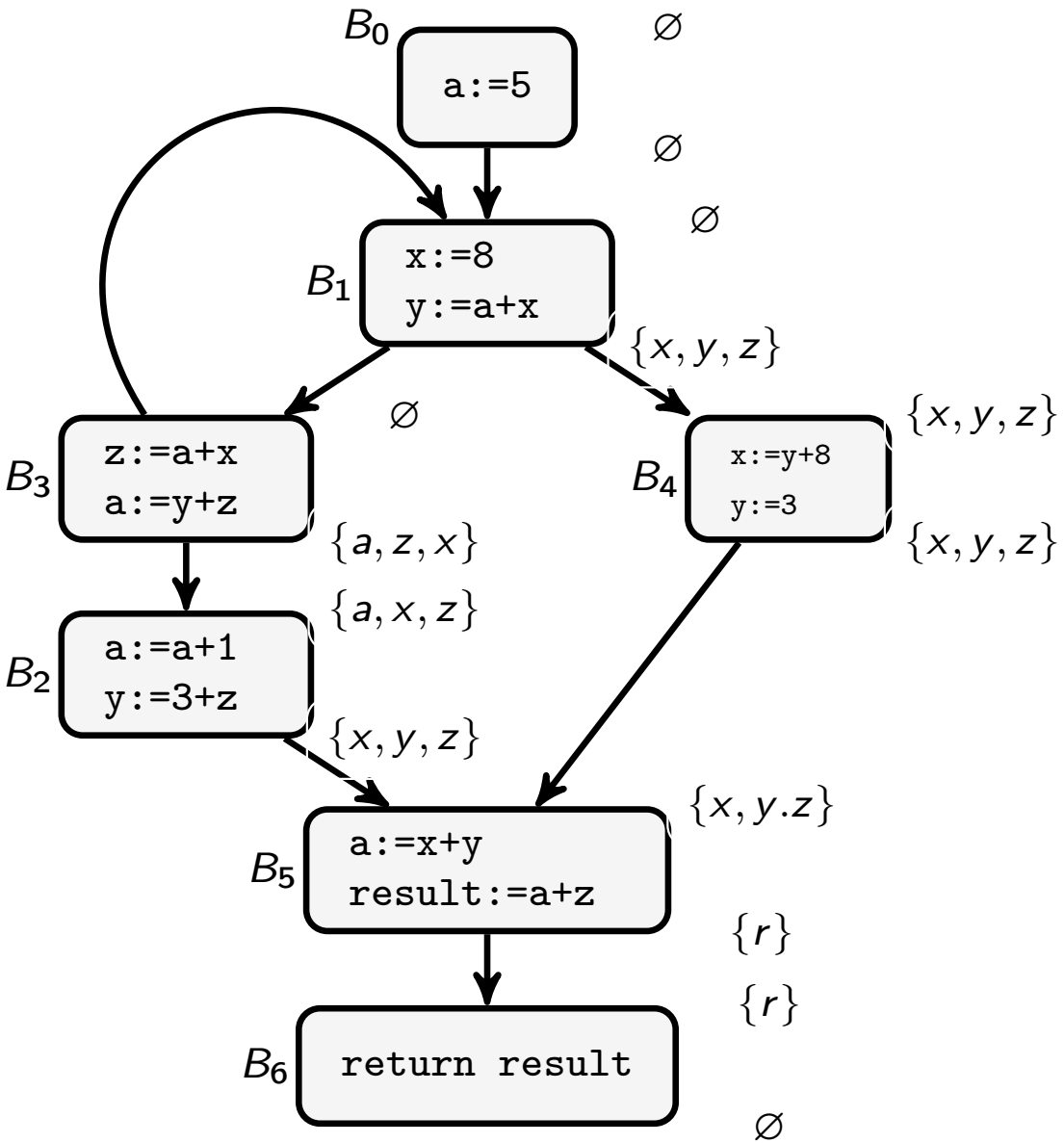
Liveness: run



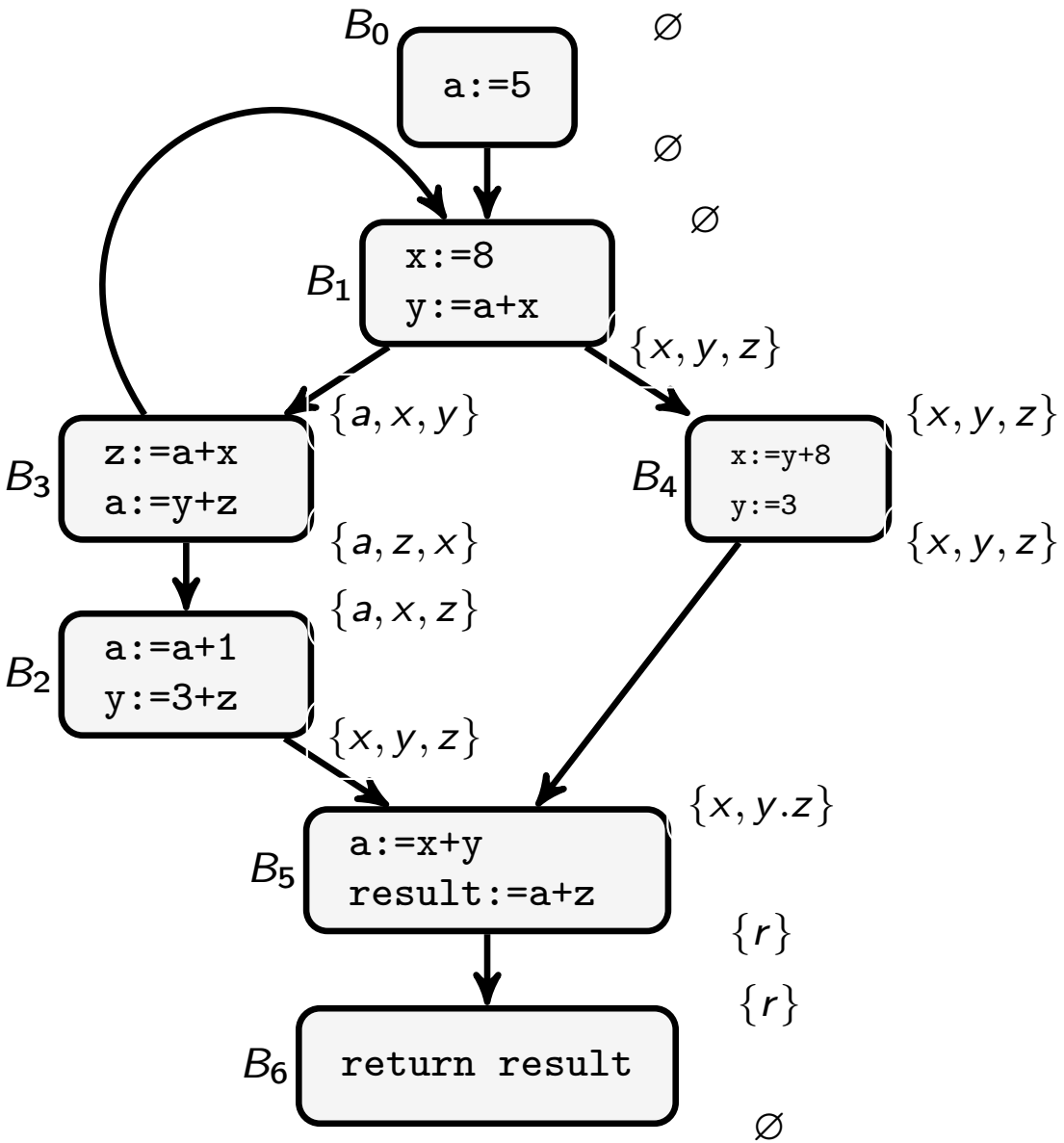
Liveness: run



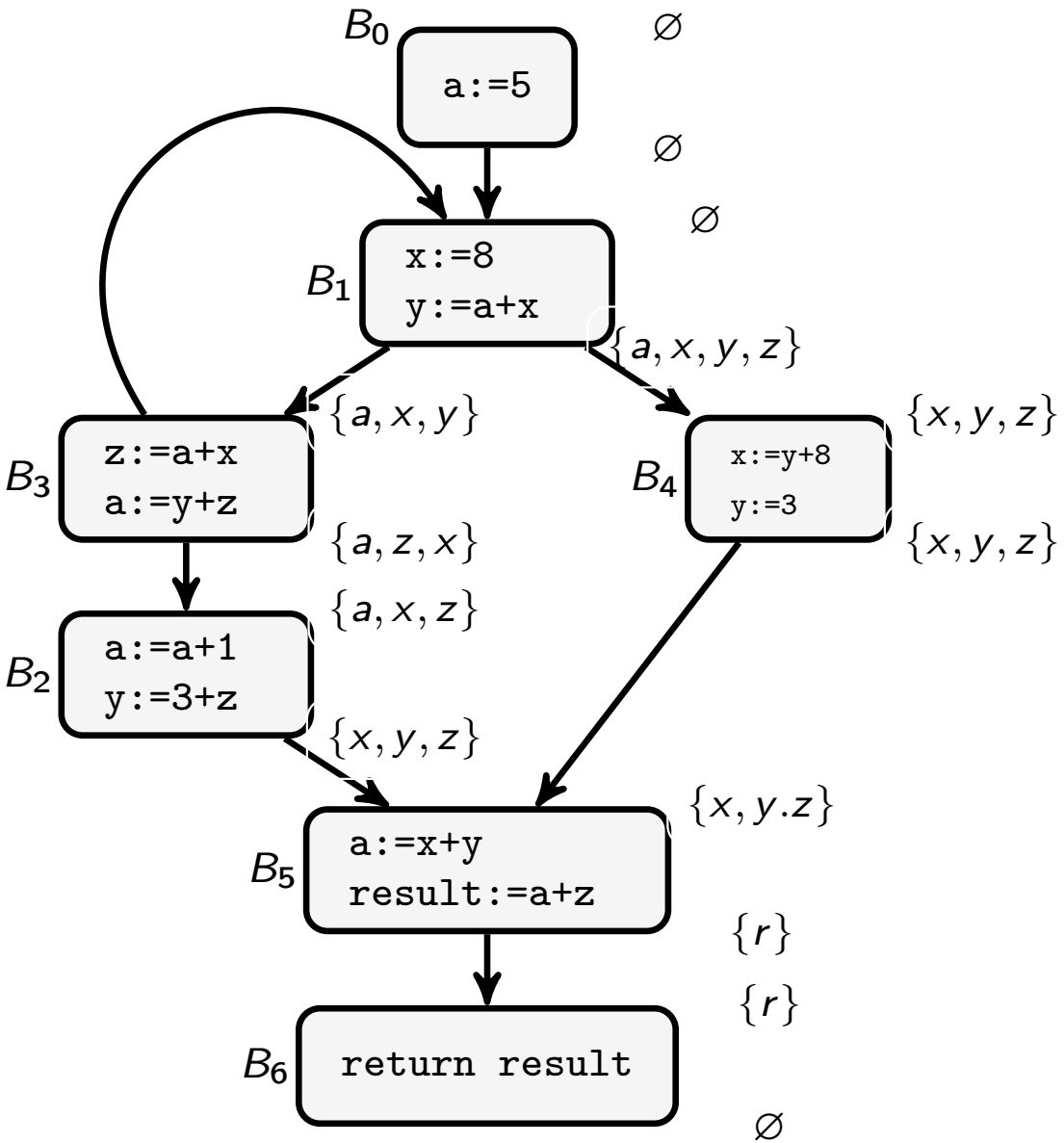
Liveness: run



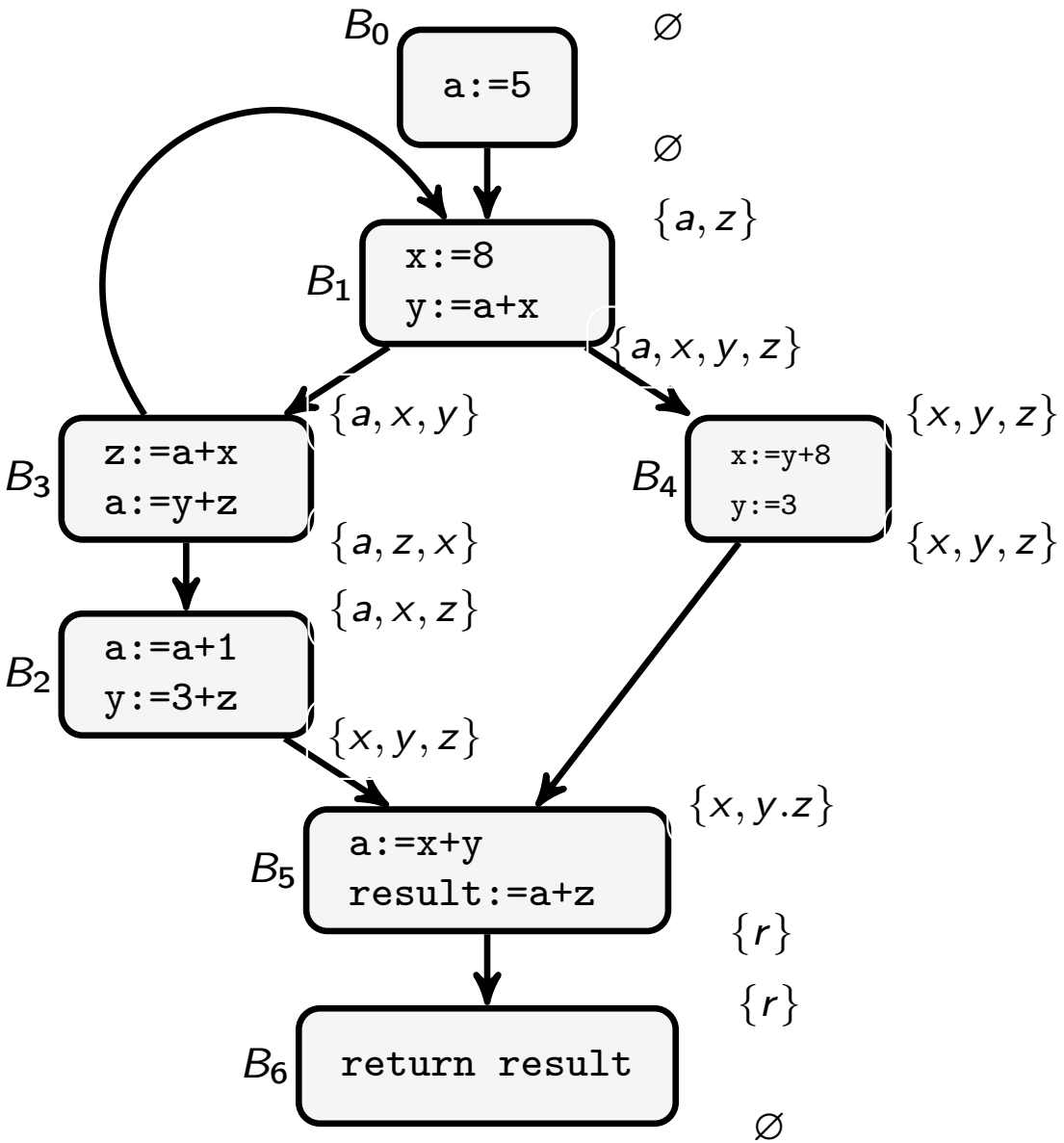
Liveness: run



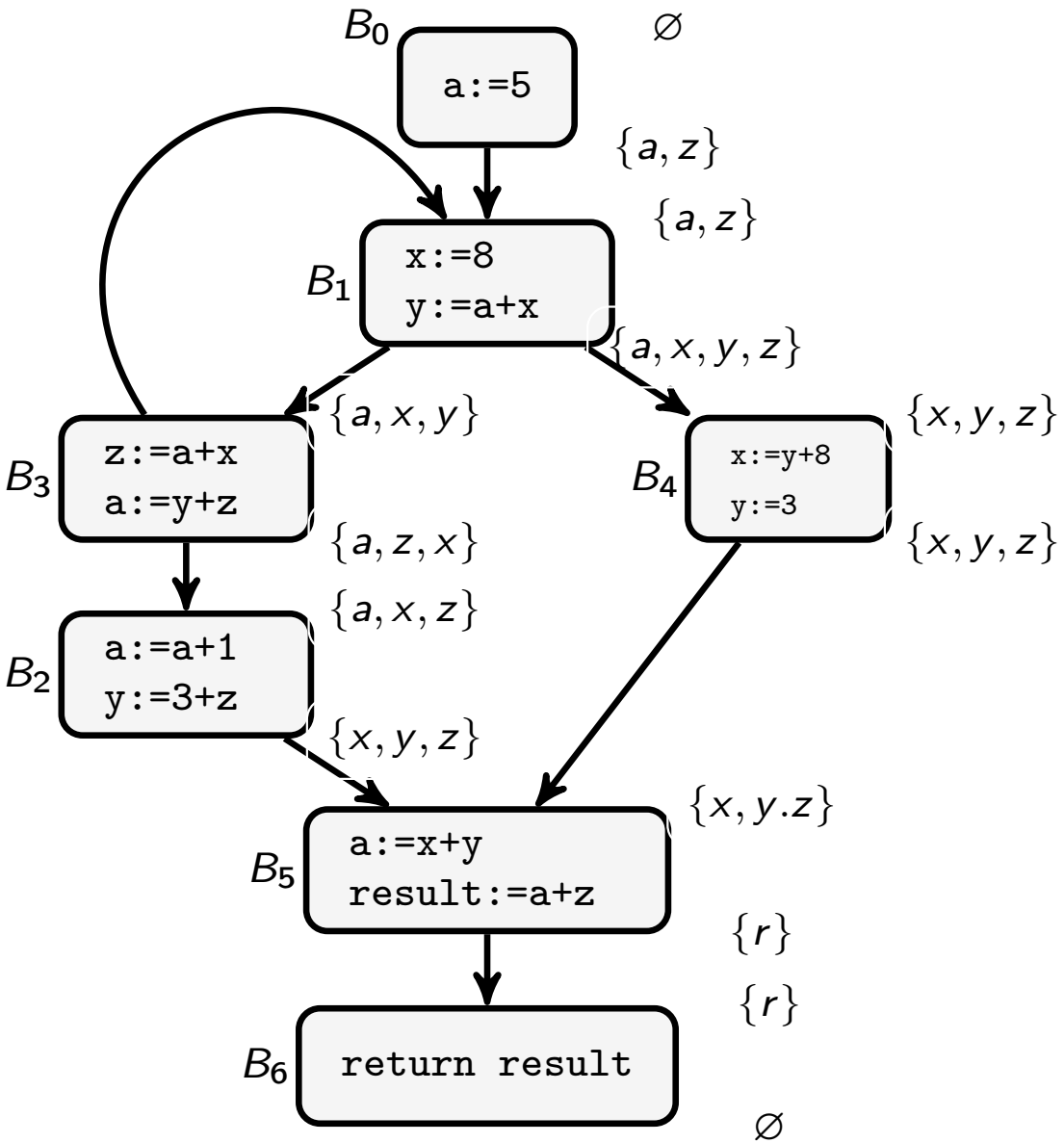
Liveness: run



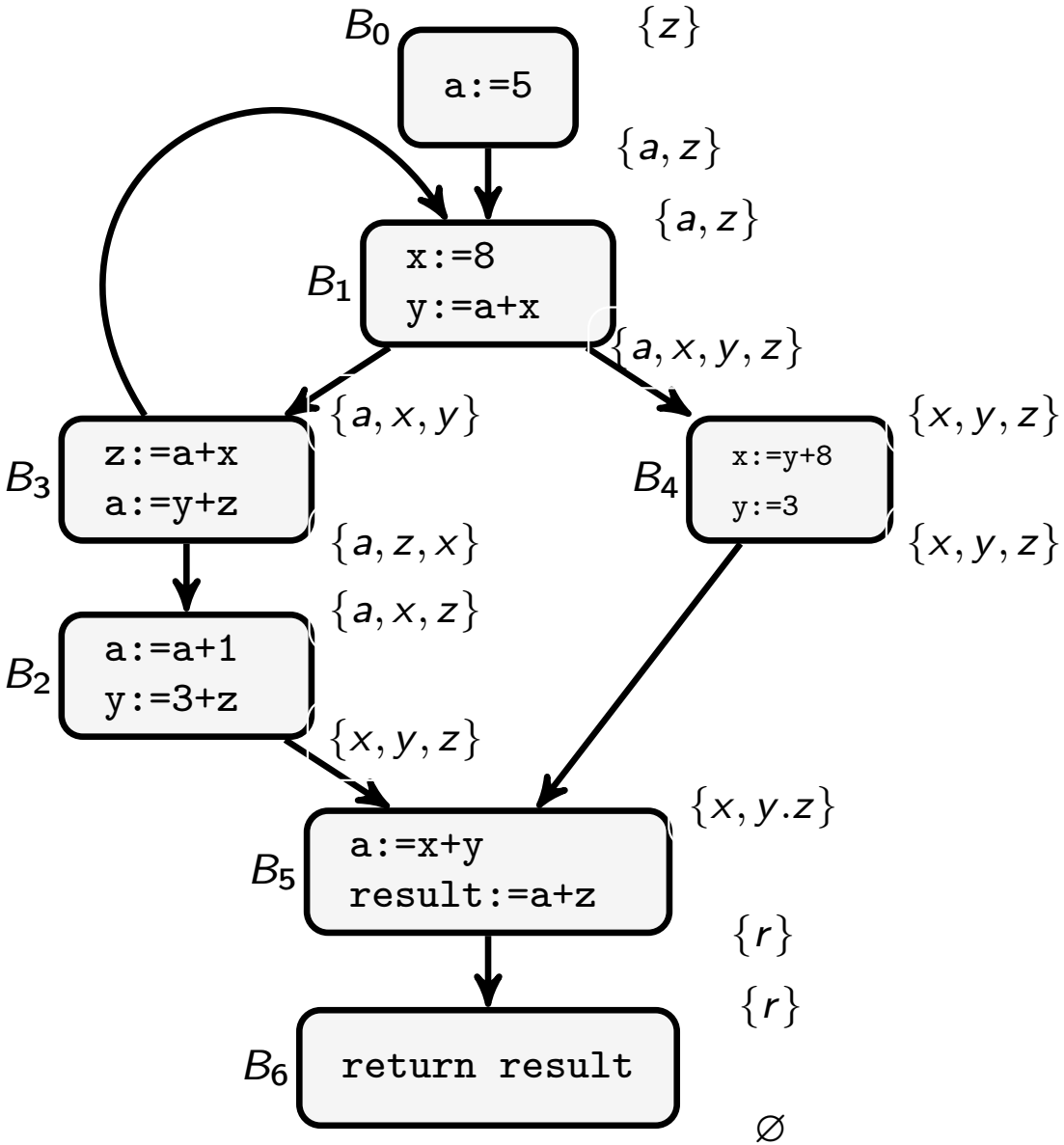
Liveness: run



Liveness: run



Liveness: run



Liveness example: remarks

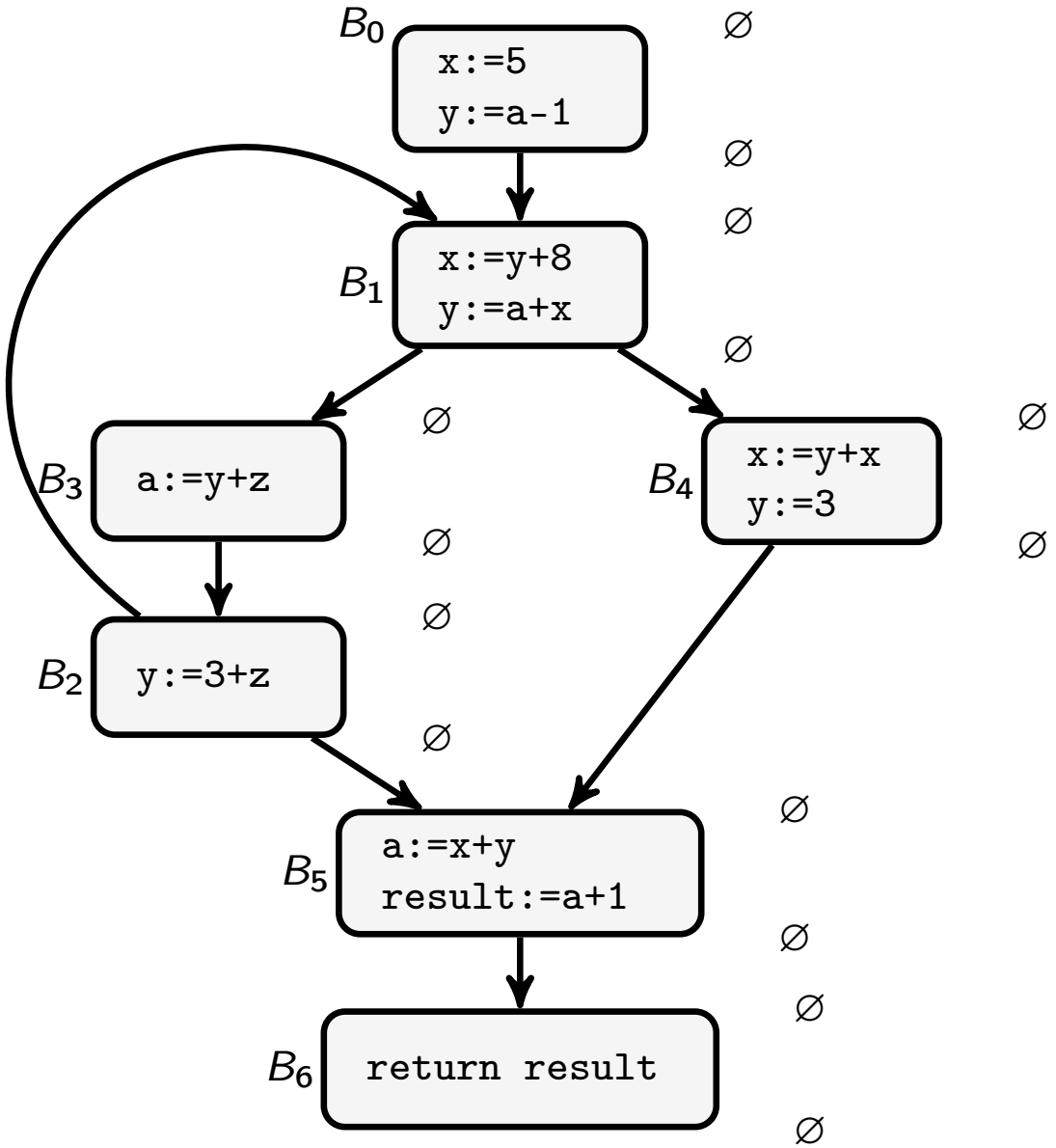
- the shown traversal strategy is (cleverly) backwards
- example resp. example run simplistic:
- the *loop* (and the choice of "evaluation" order):

"harmless loop"

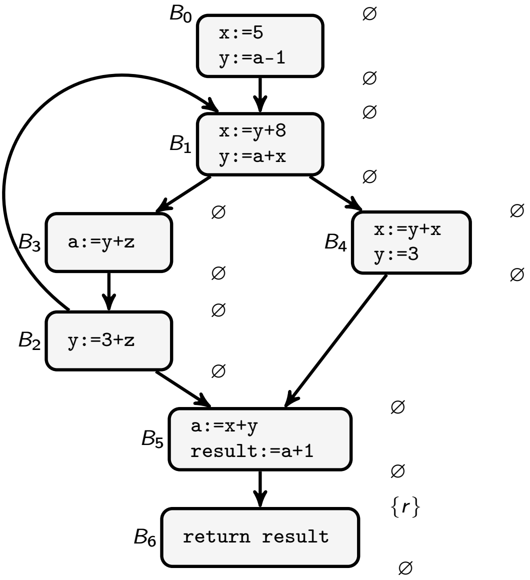
after having updated the *outLive* info for B_1 following the edge from B_3 to B_1 *backwards* (propagating flow from B_1 back to B_3) does not increase the current solution for B_3

- no need (in this particular order) for continuing the iterative search for stabilization
- in other examples: loop iteration cannot be avoided
- note also: end result (after stabilization) independent from evaluation order! (only some strategies may stabilize faster...)

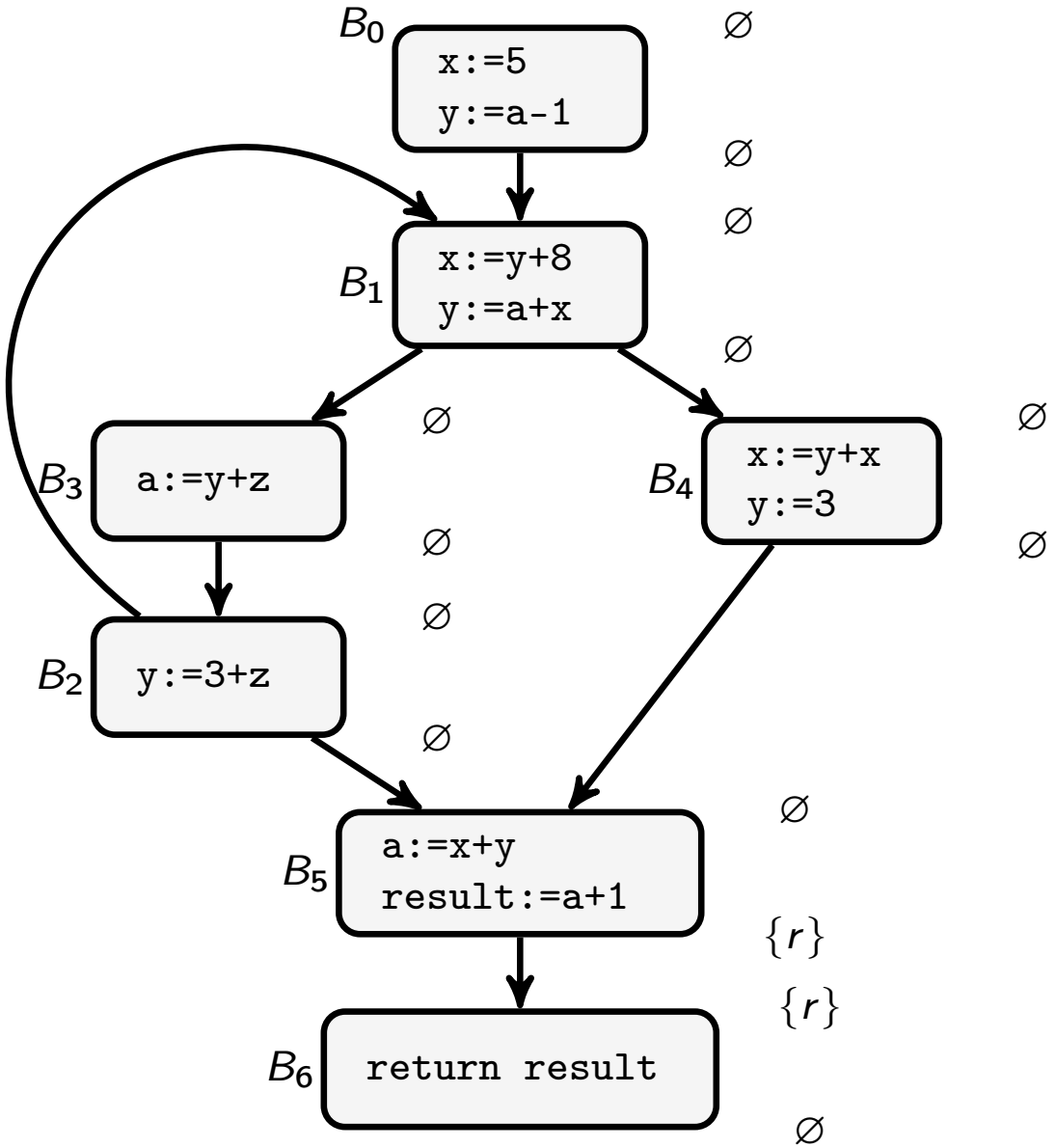
Another example



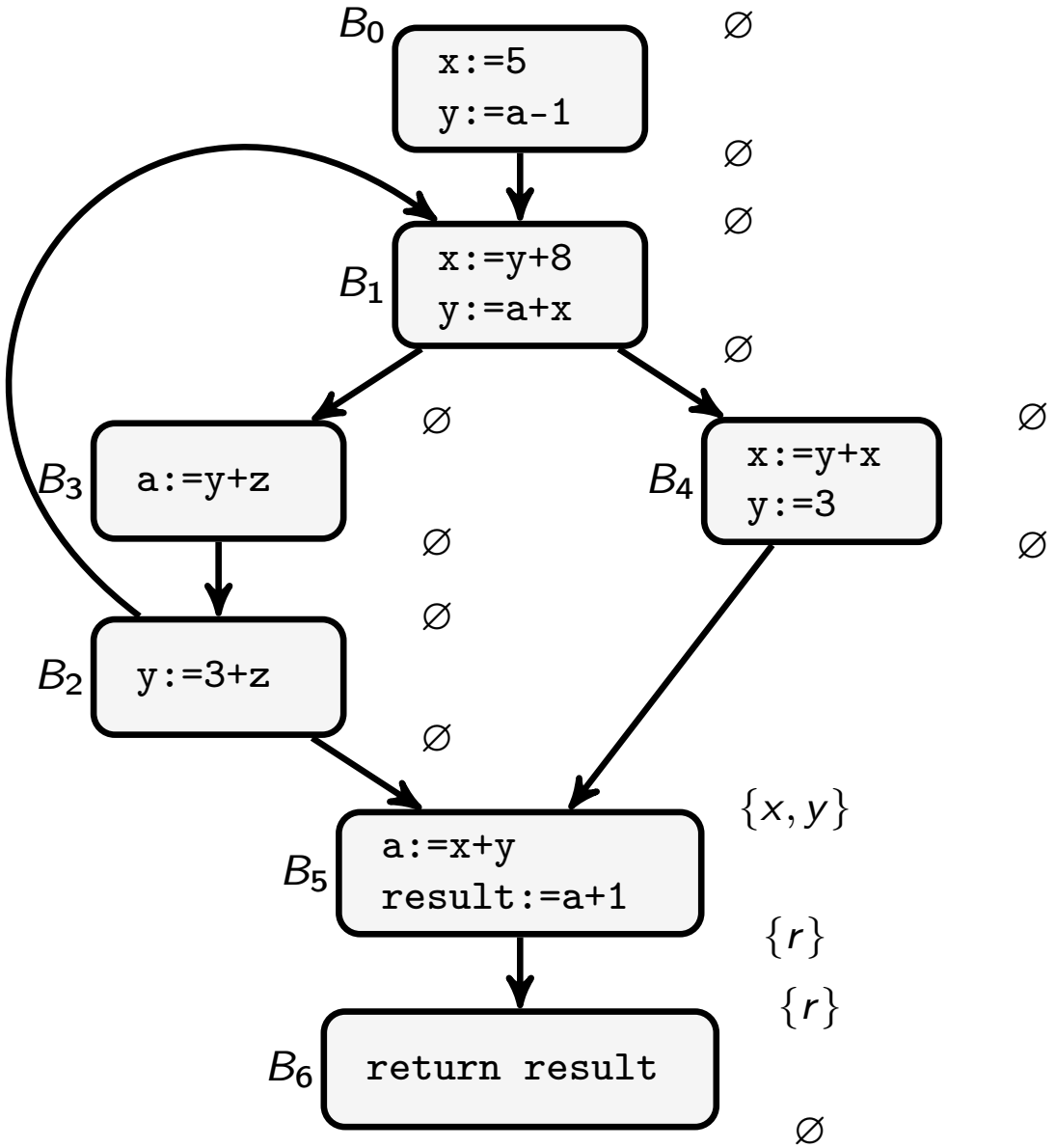
Another example



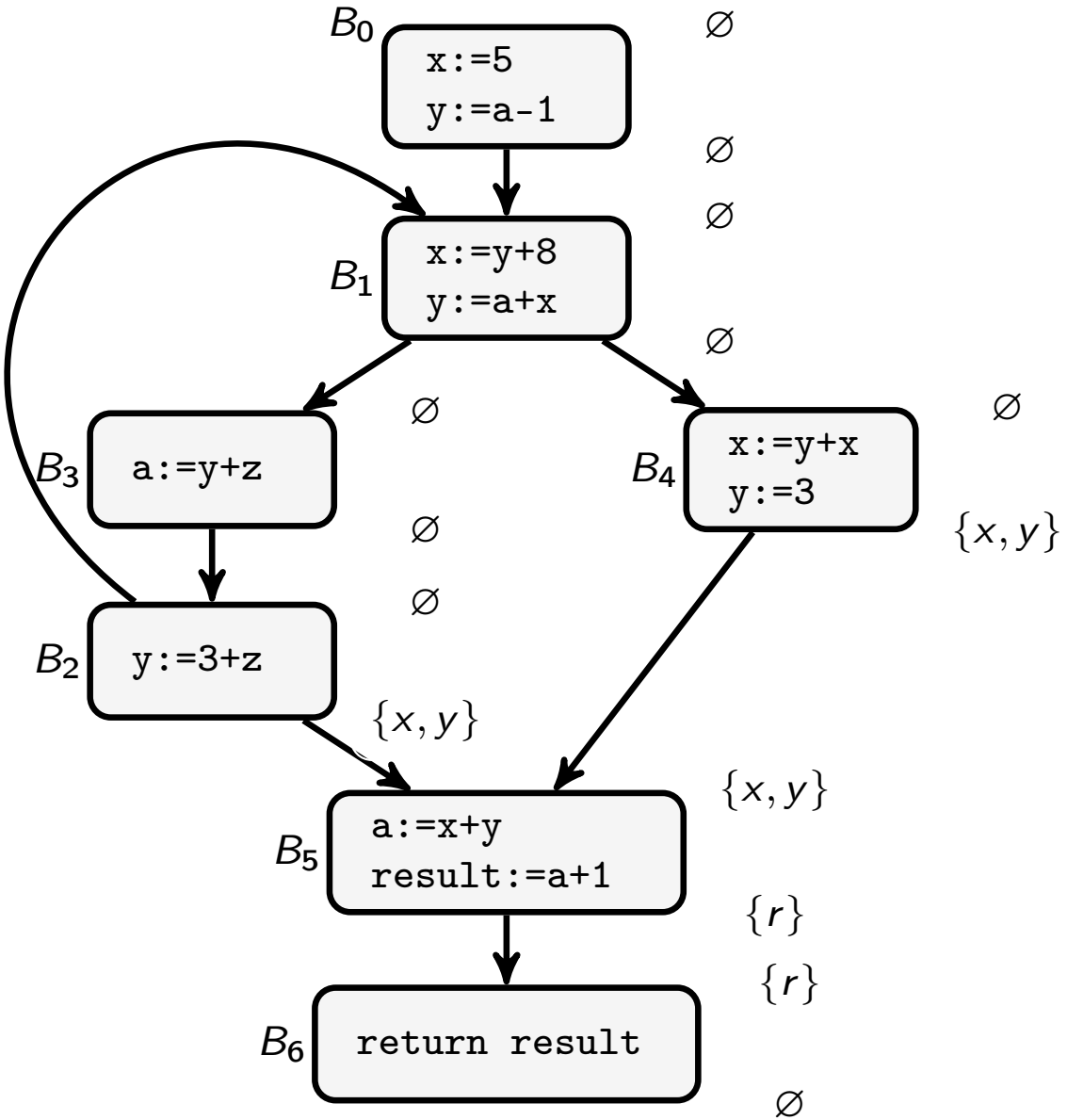
Another example



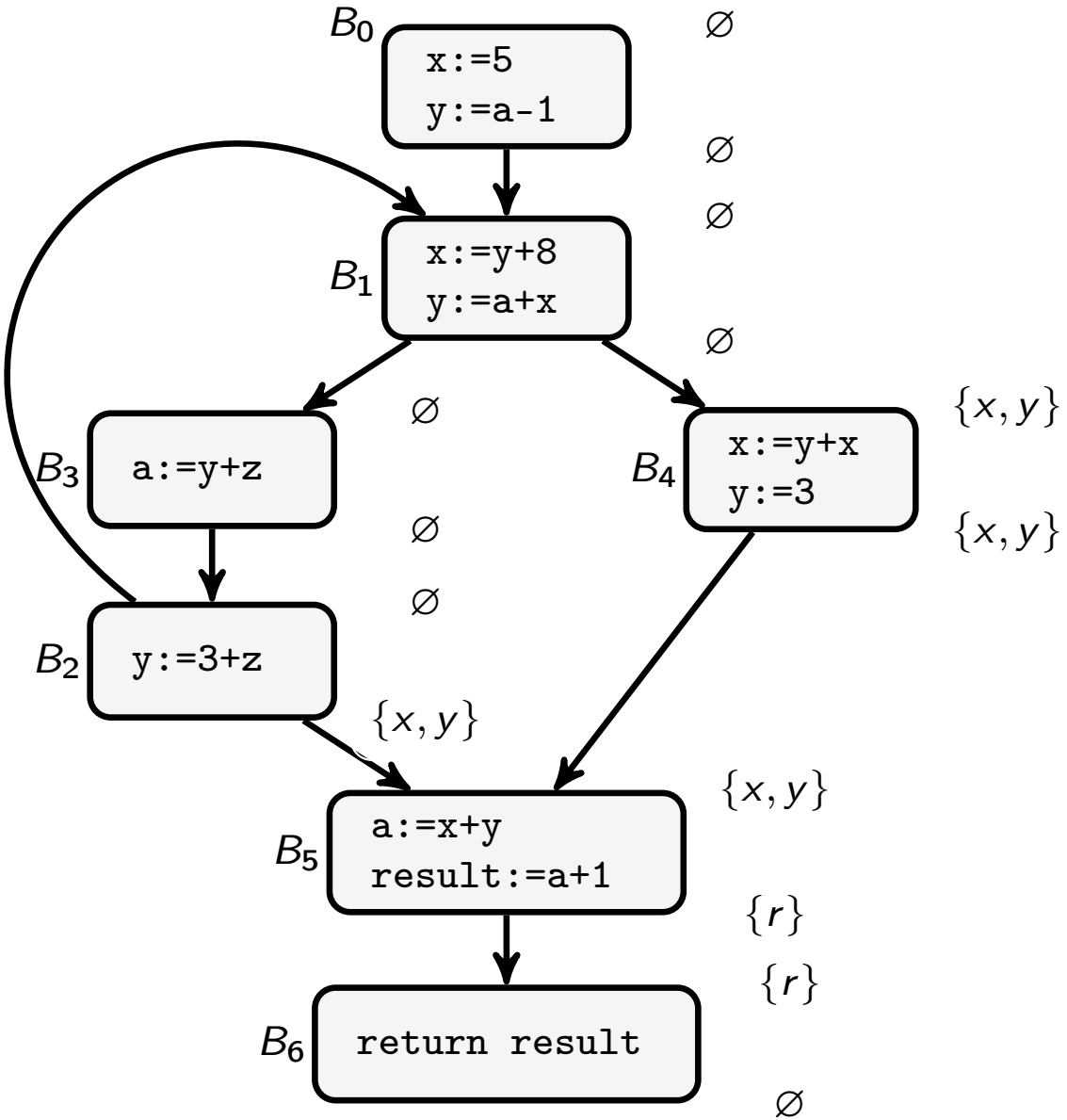
Another example



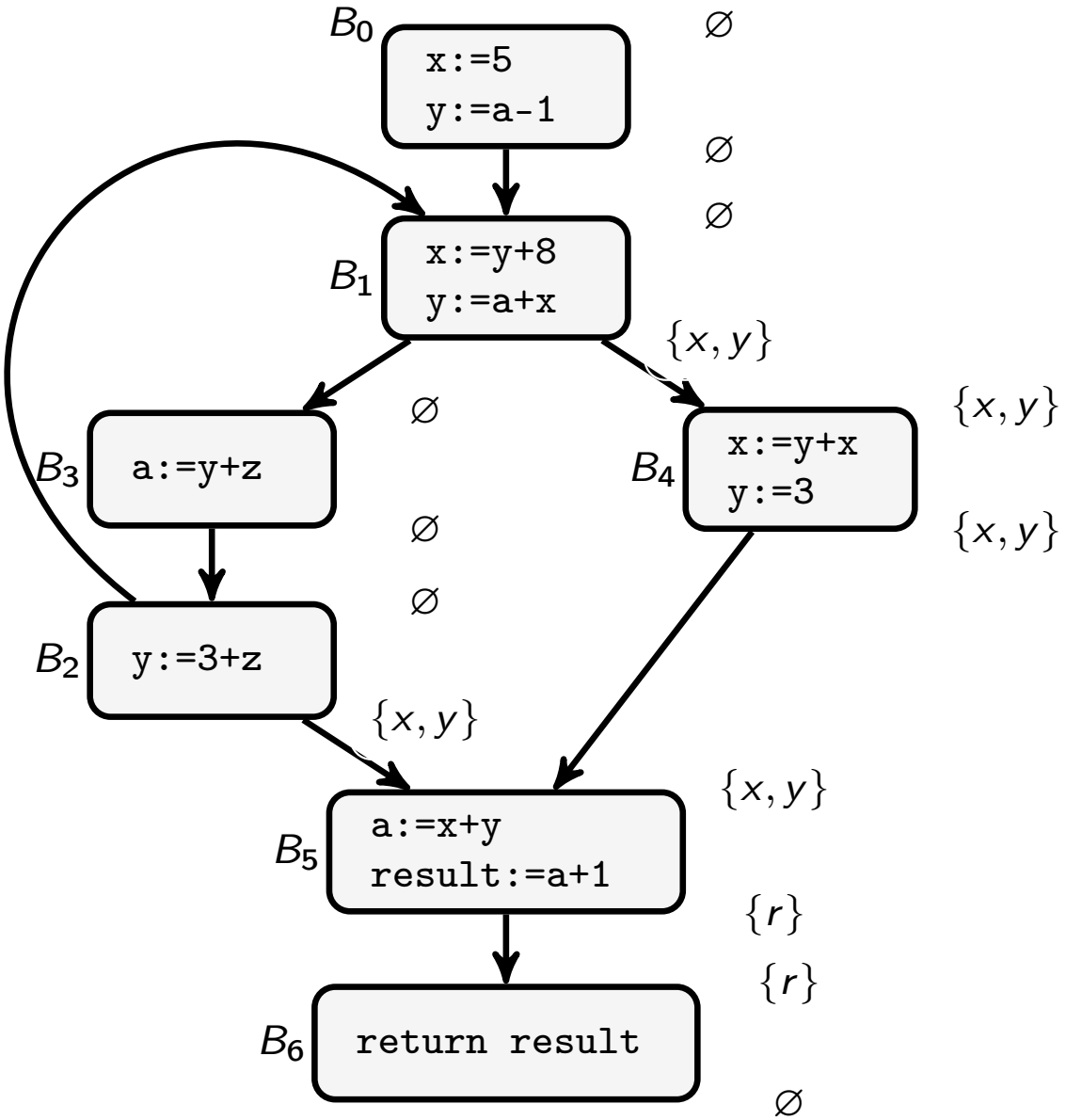
Another example



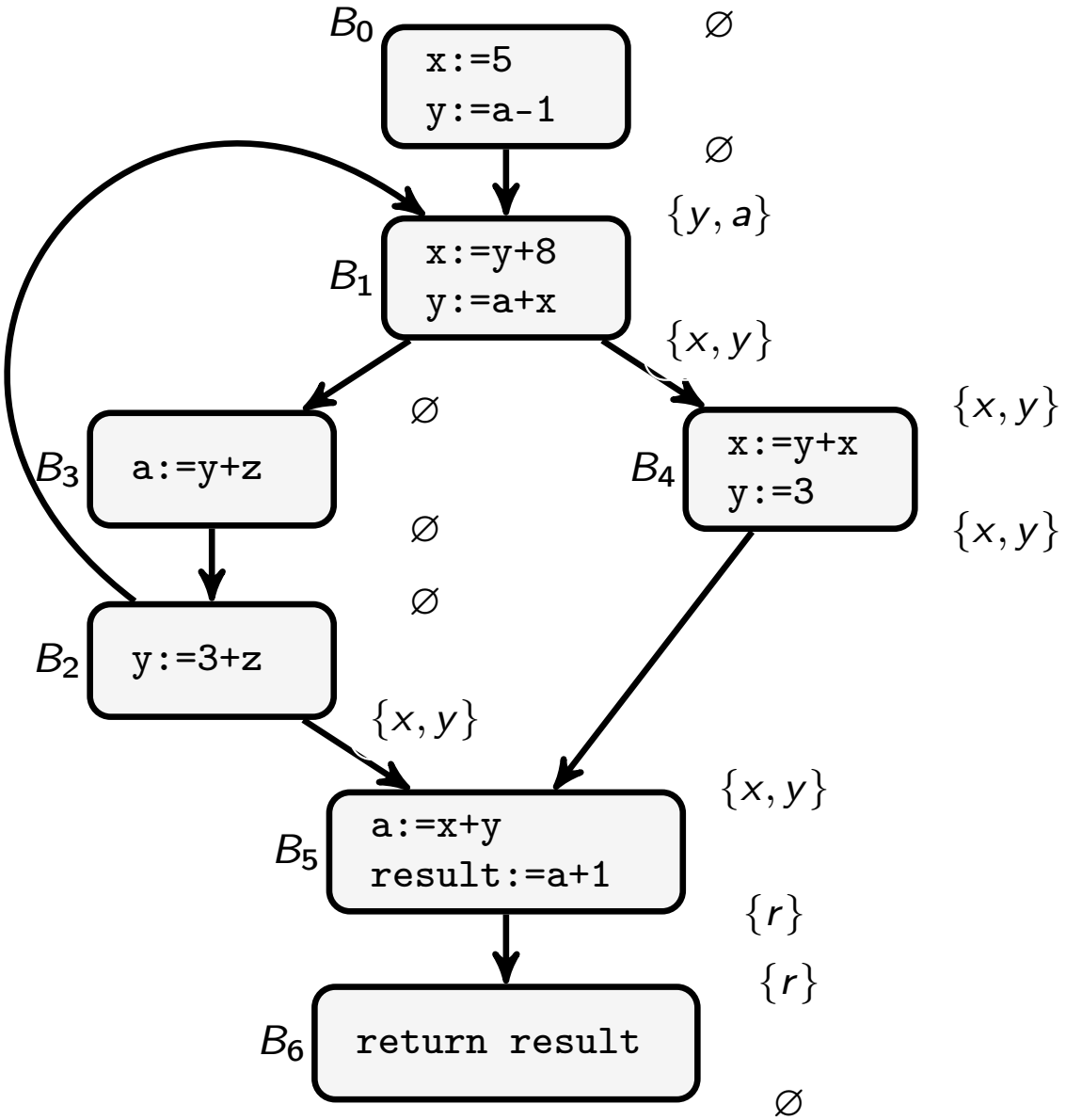
Another example



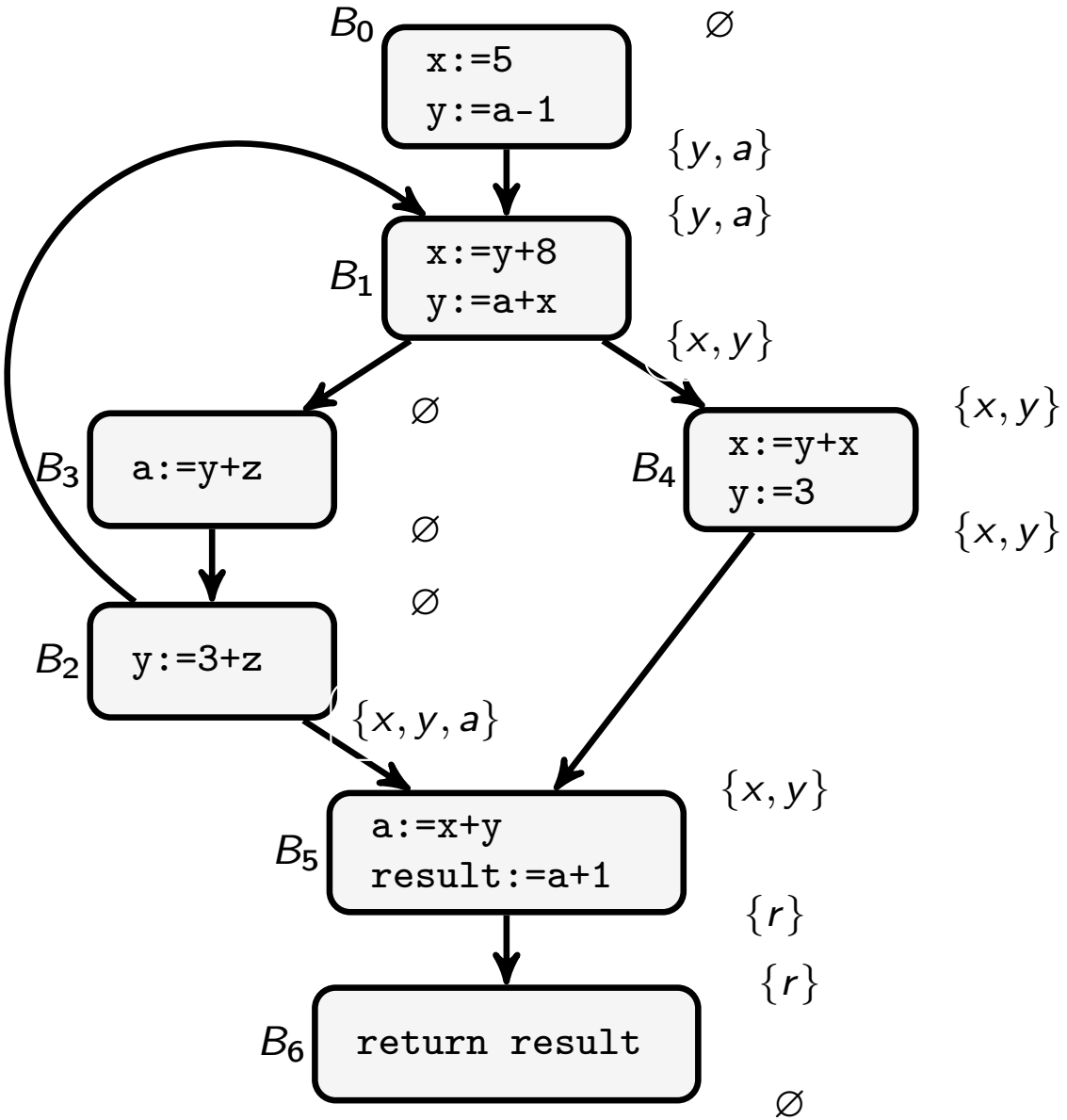
Another example



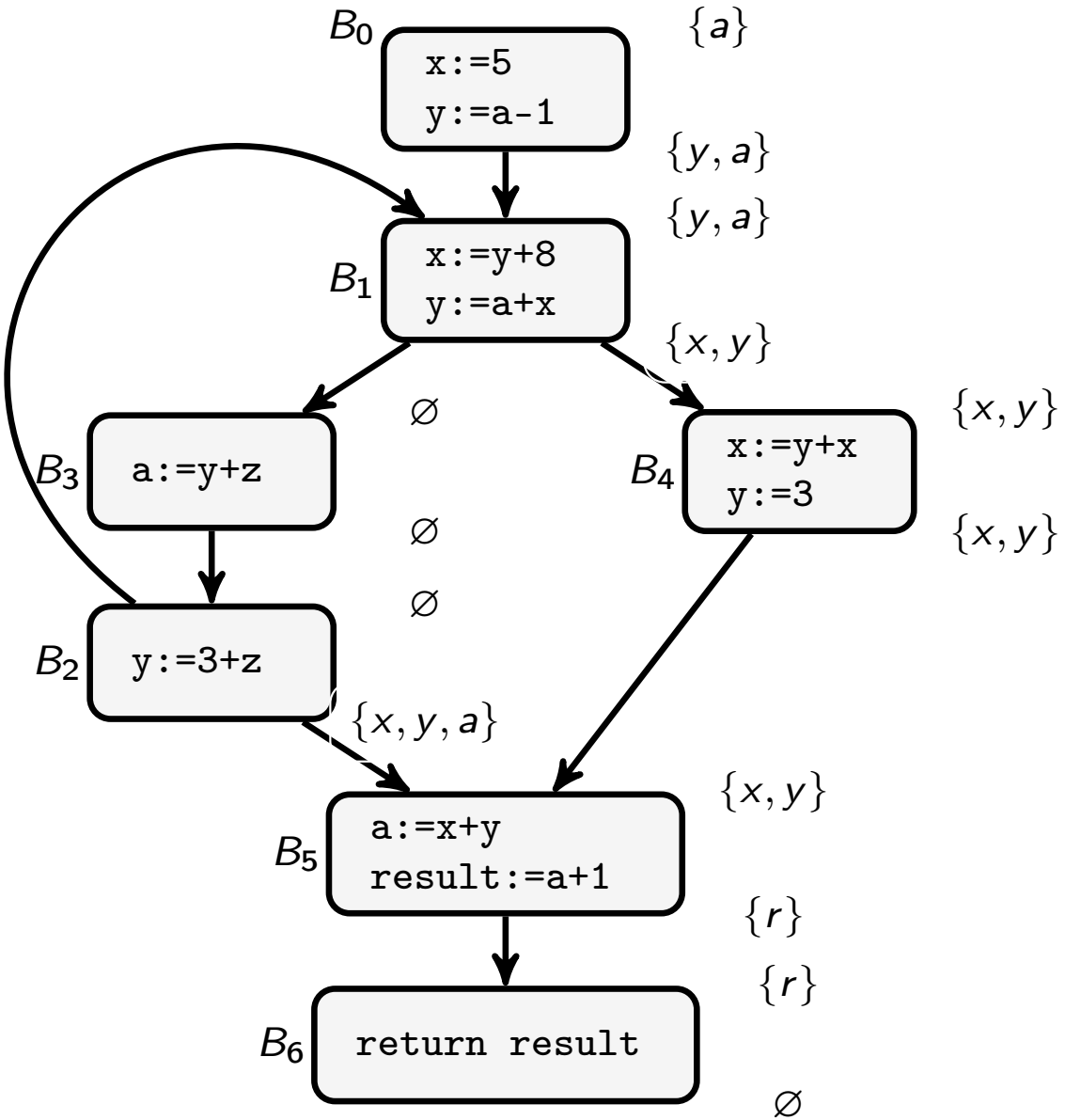
Another example



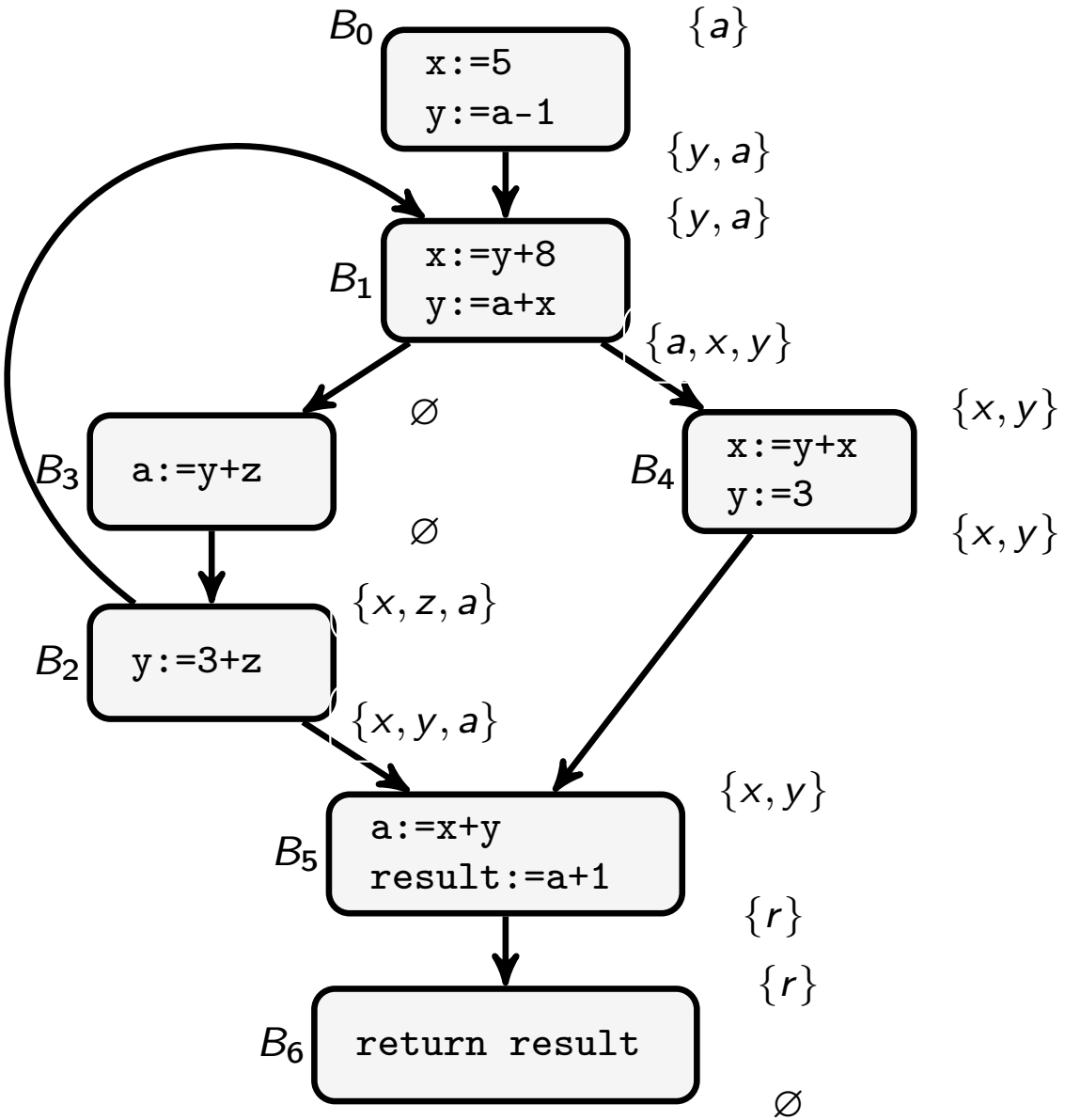
Another example



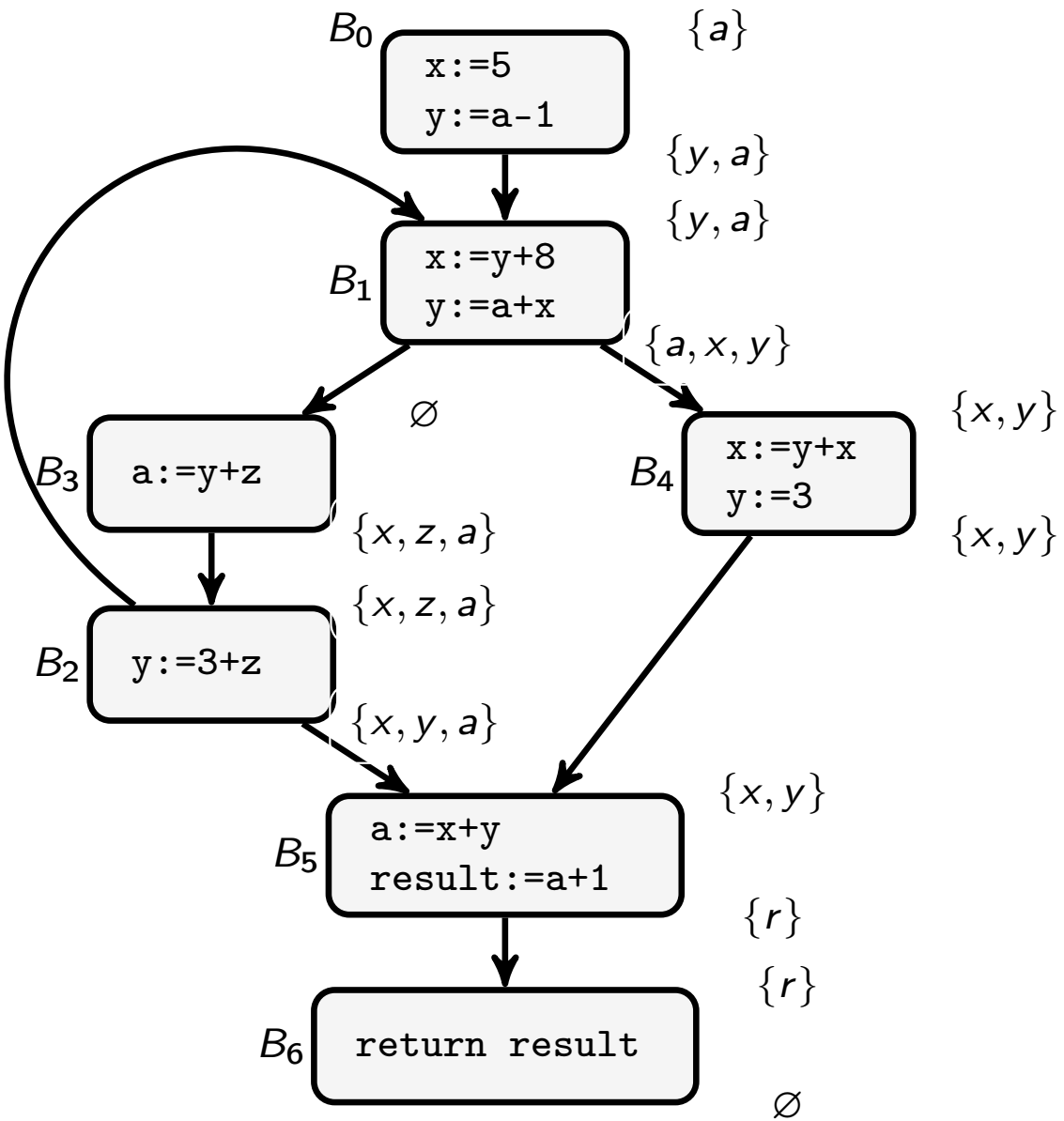
Another example



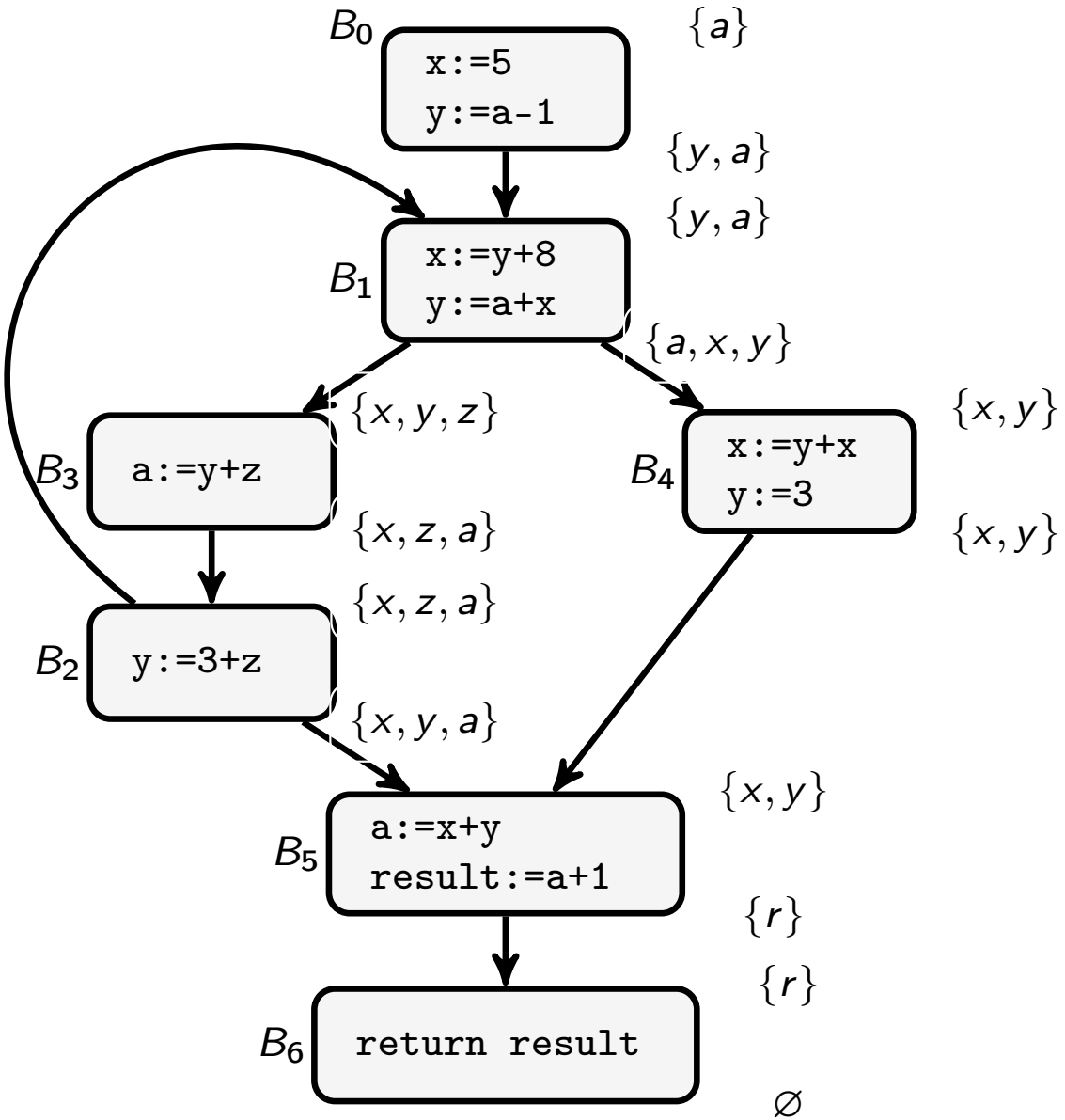
Another example



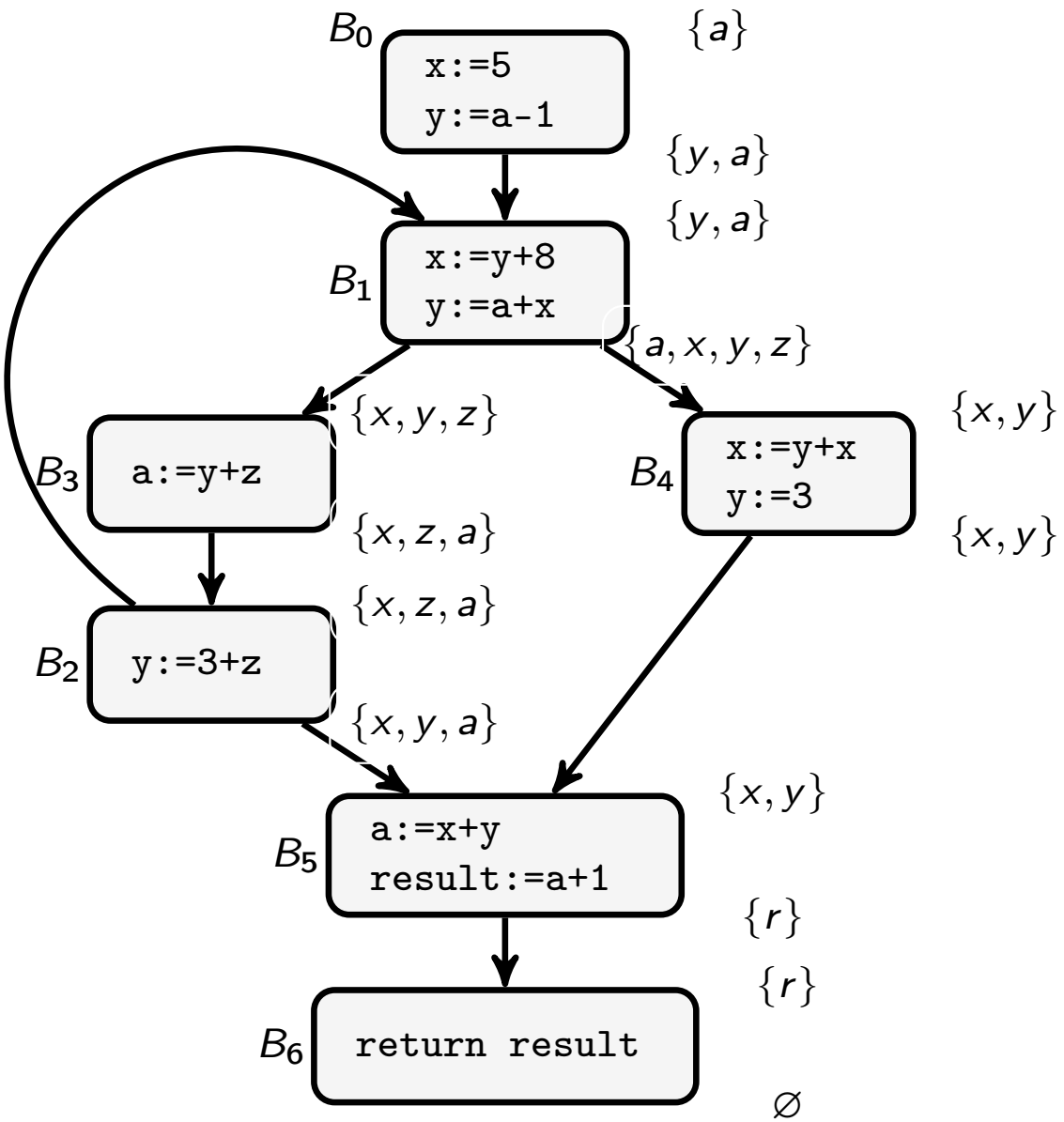
Another example



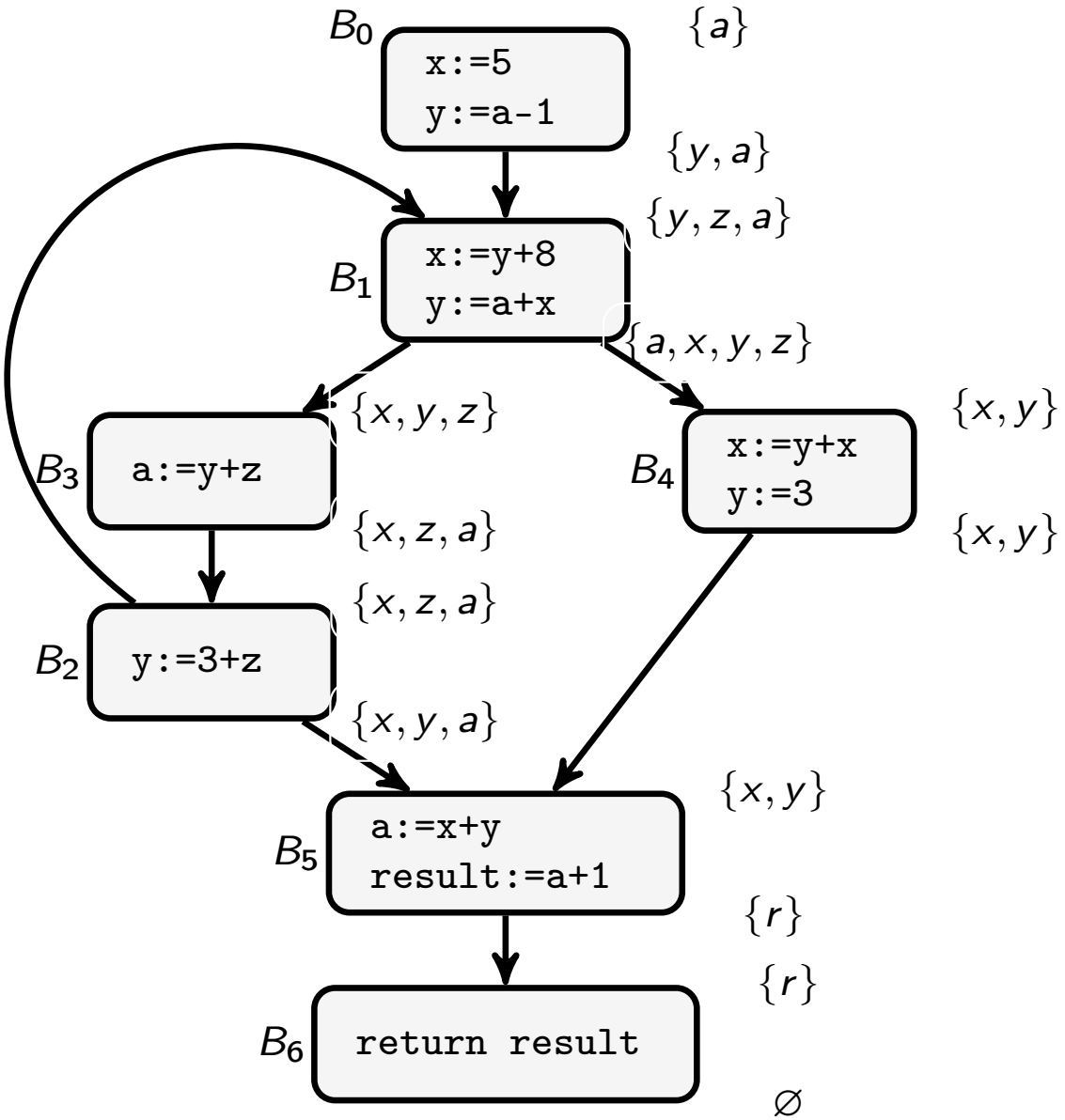
Another example



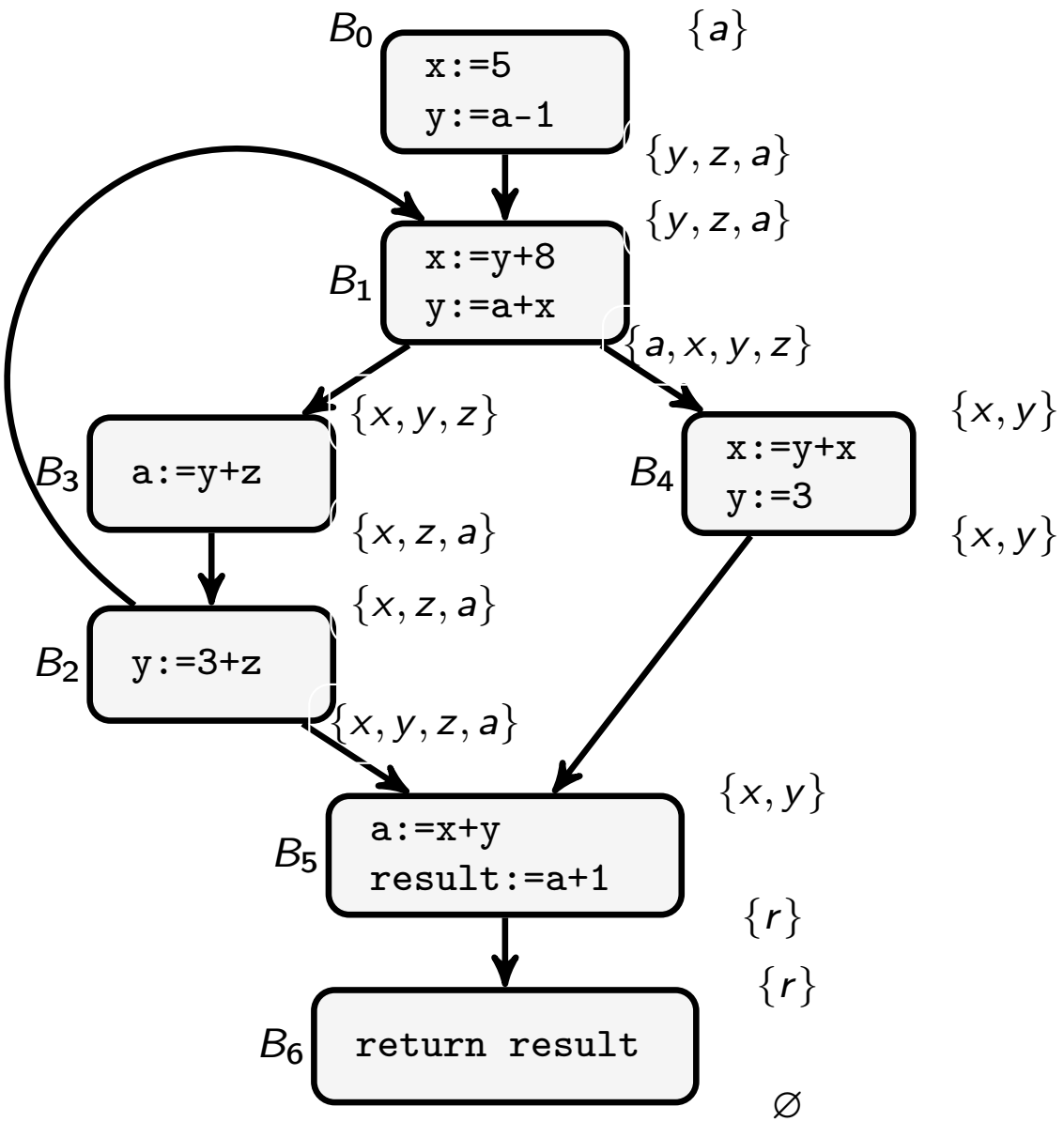
Another example



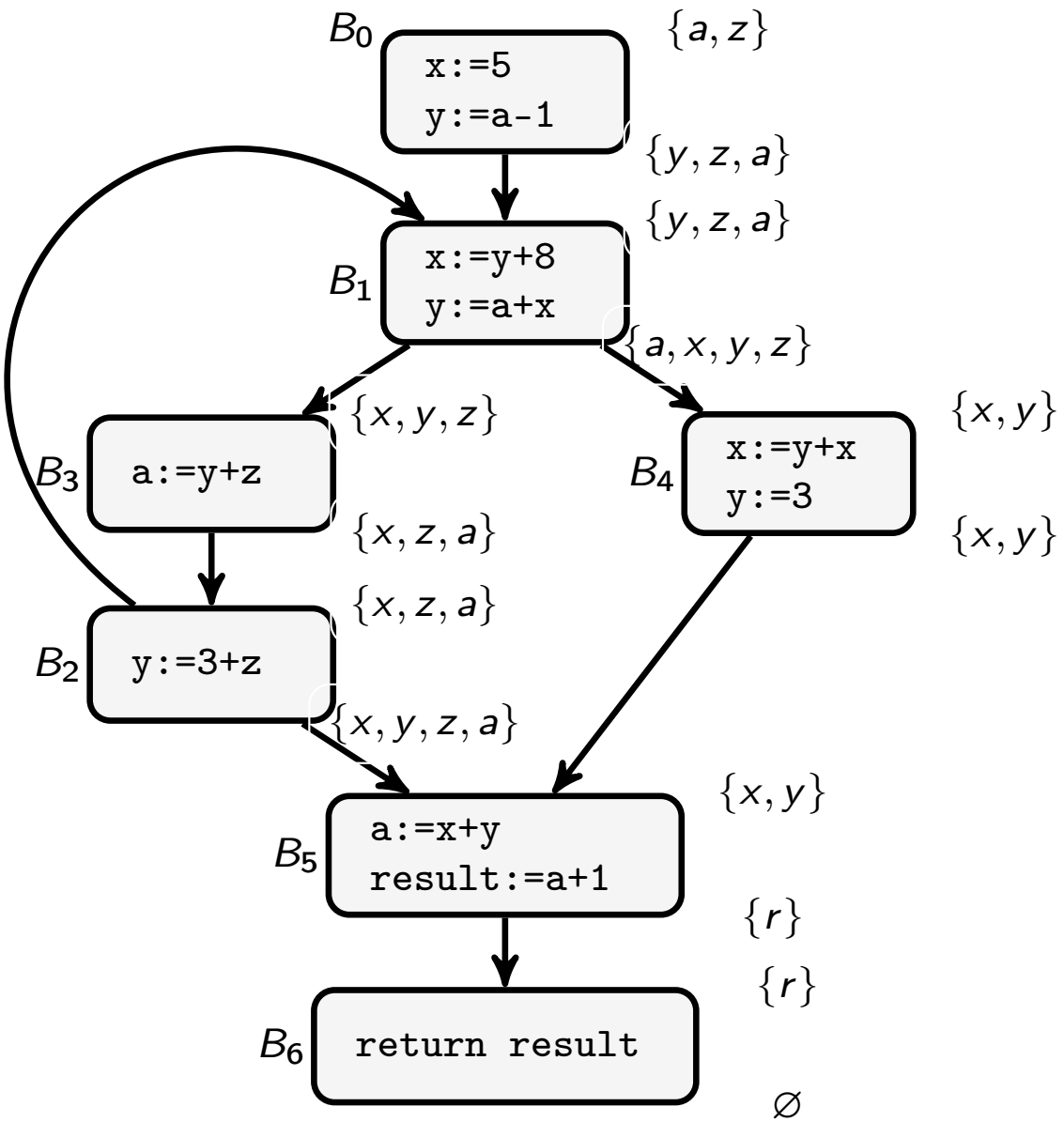
Another example



Another example



Another example



Example remarks

- loop: this time leads to updating estimation more than once
- evaluation order not chose ideally

Precomputing the block-local "liveness effects"

- *precomputation* of the relevant info: efficiency
- traditionally: represented as *kill* and *generate* information
- here (for liveness)
  1. kill: variable instances, which are overwritten
  2. generate: variables used in the block (before overwritten)
  3. rests: all other variables won't change their status

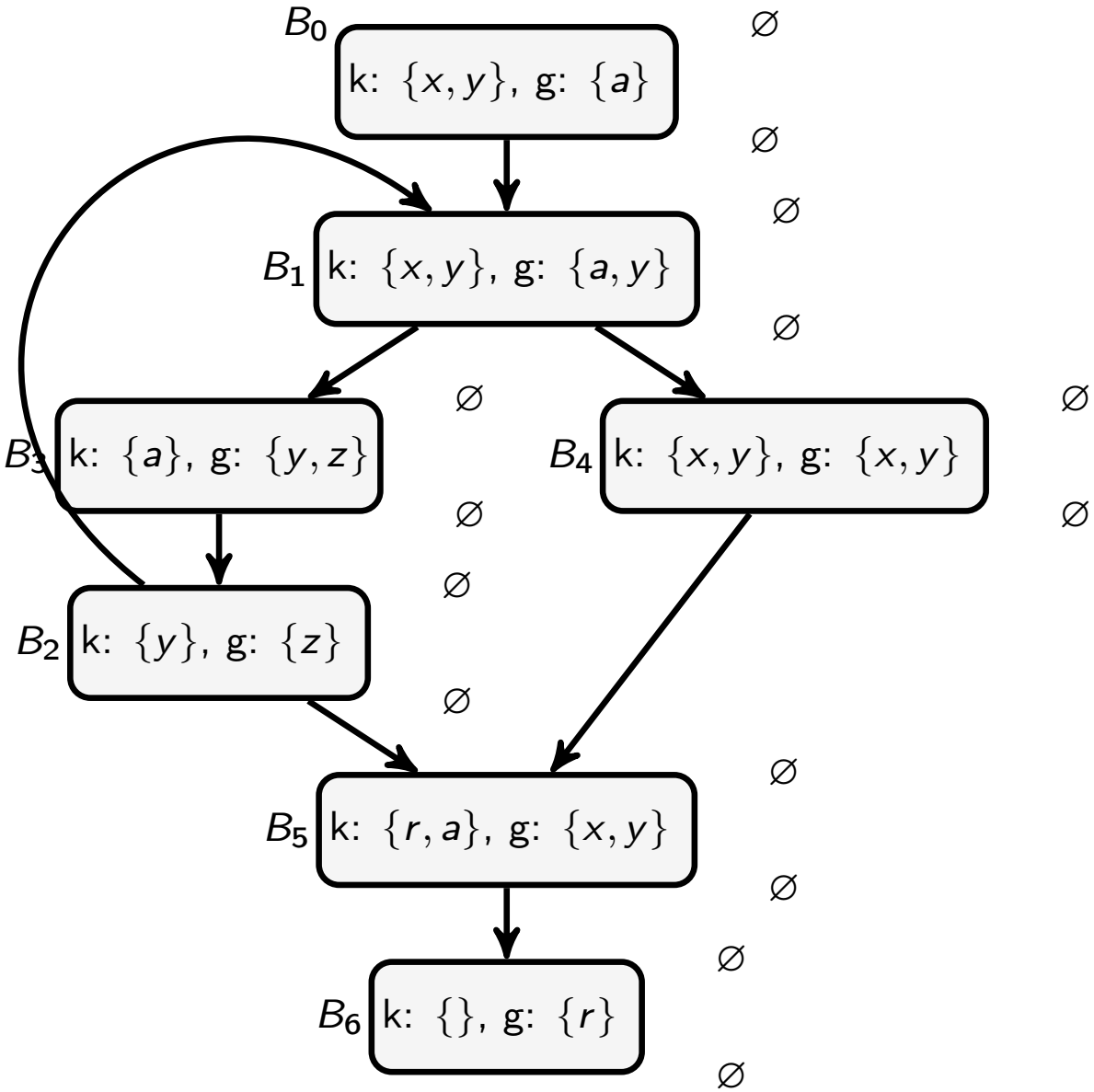
Constraint per basic block (transfer function)

$$inLive = outLive \backslash kill(B) \cup generate(B)$$

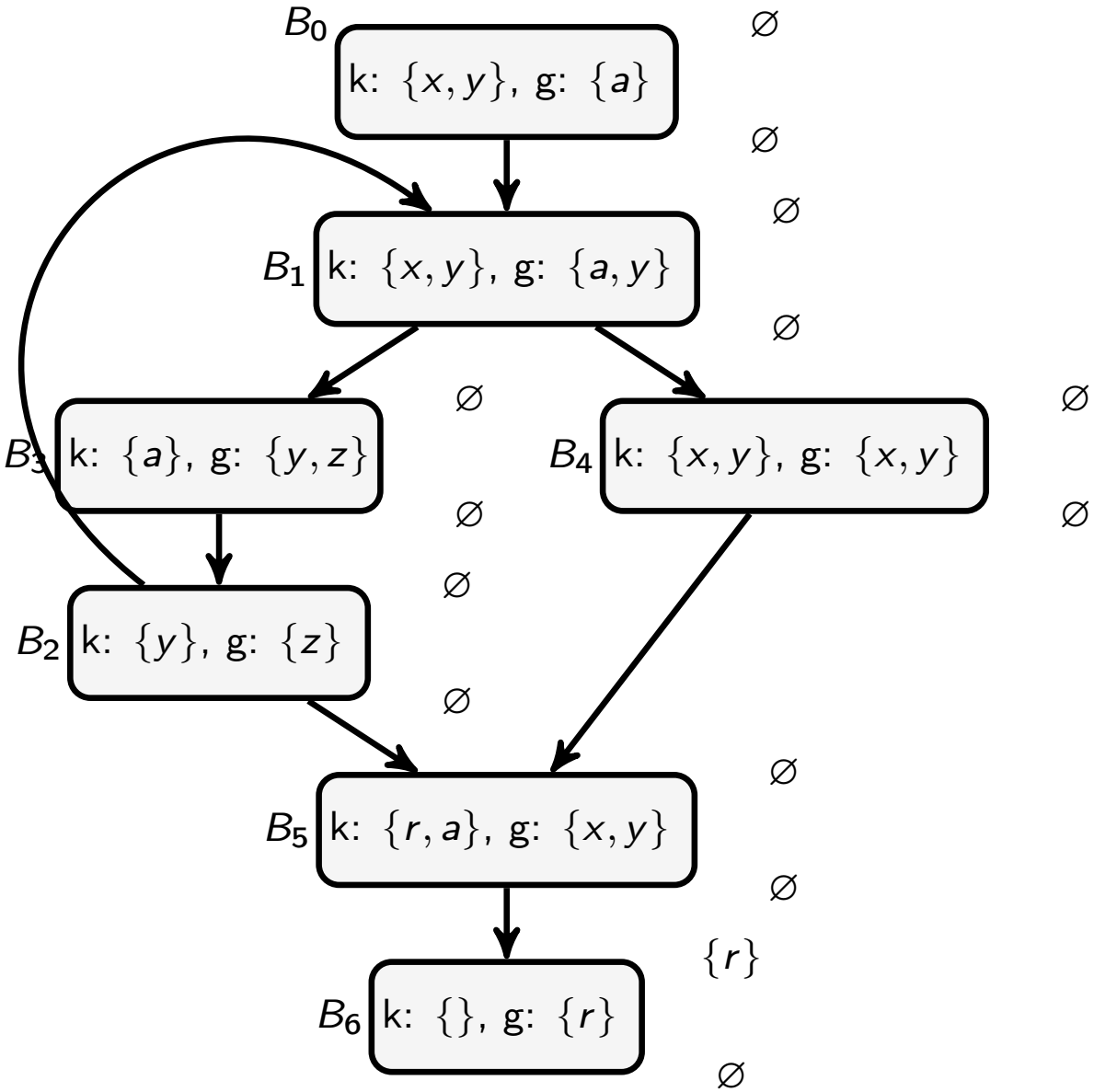
- note:
  - order of kill and generate in above's equation[13]
  - a variable killed in a block may be "revived" in a block
- simplest (one line) example: `x := x +1`

[13] In principle, one could also arrange the opposite order (interpreting kill and generatate slightly differently). One can also define the so-called transfer function directly, without splitting into kill and generate. The phrasing using such tranfer functions: works for other DFA as well.

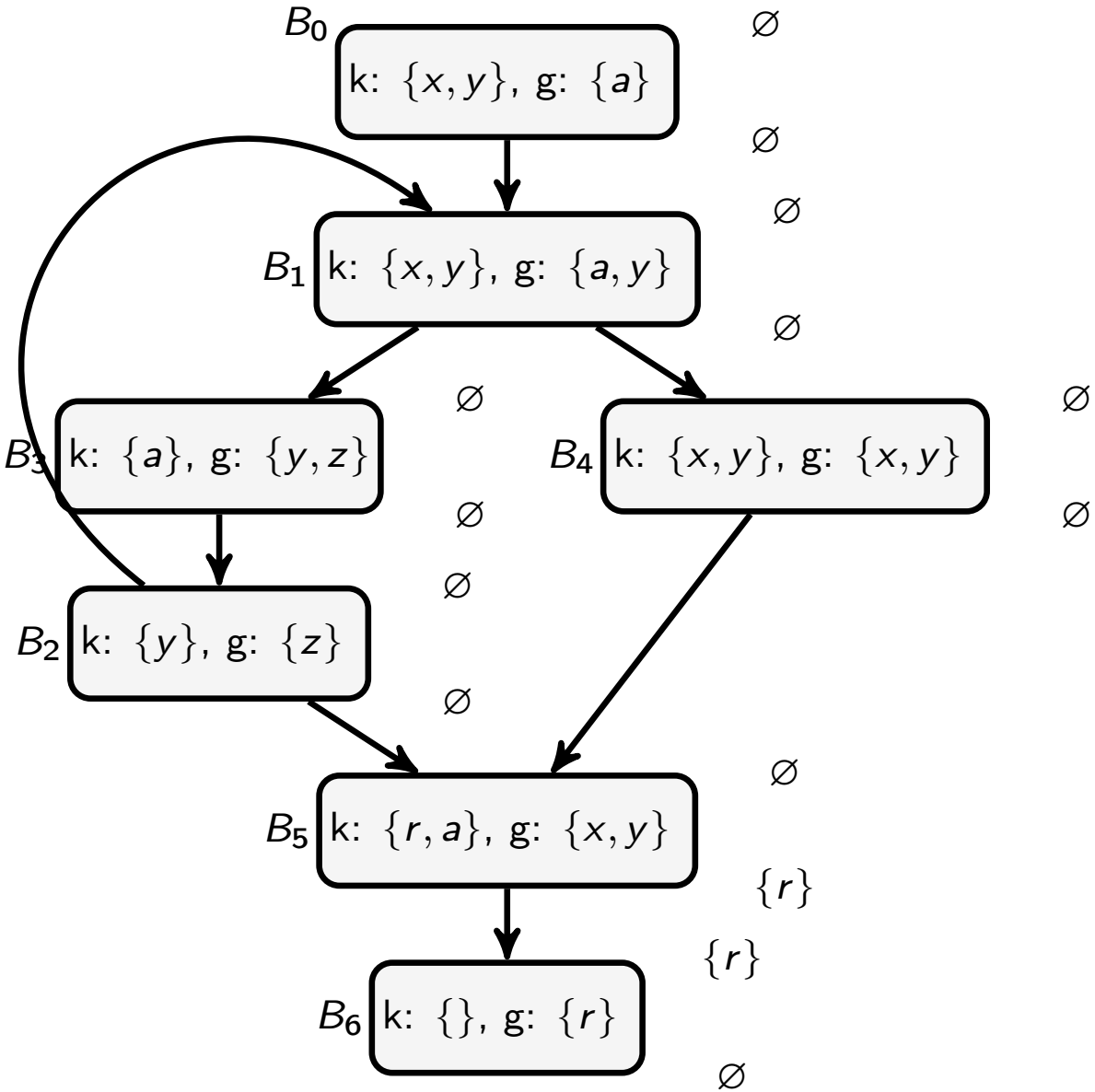
Example once again



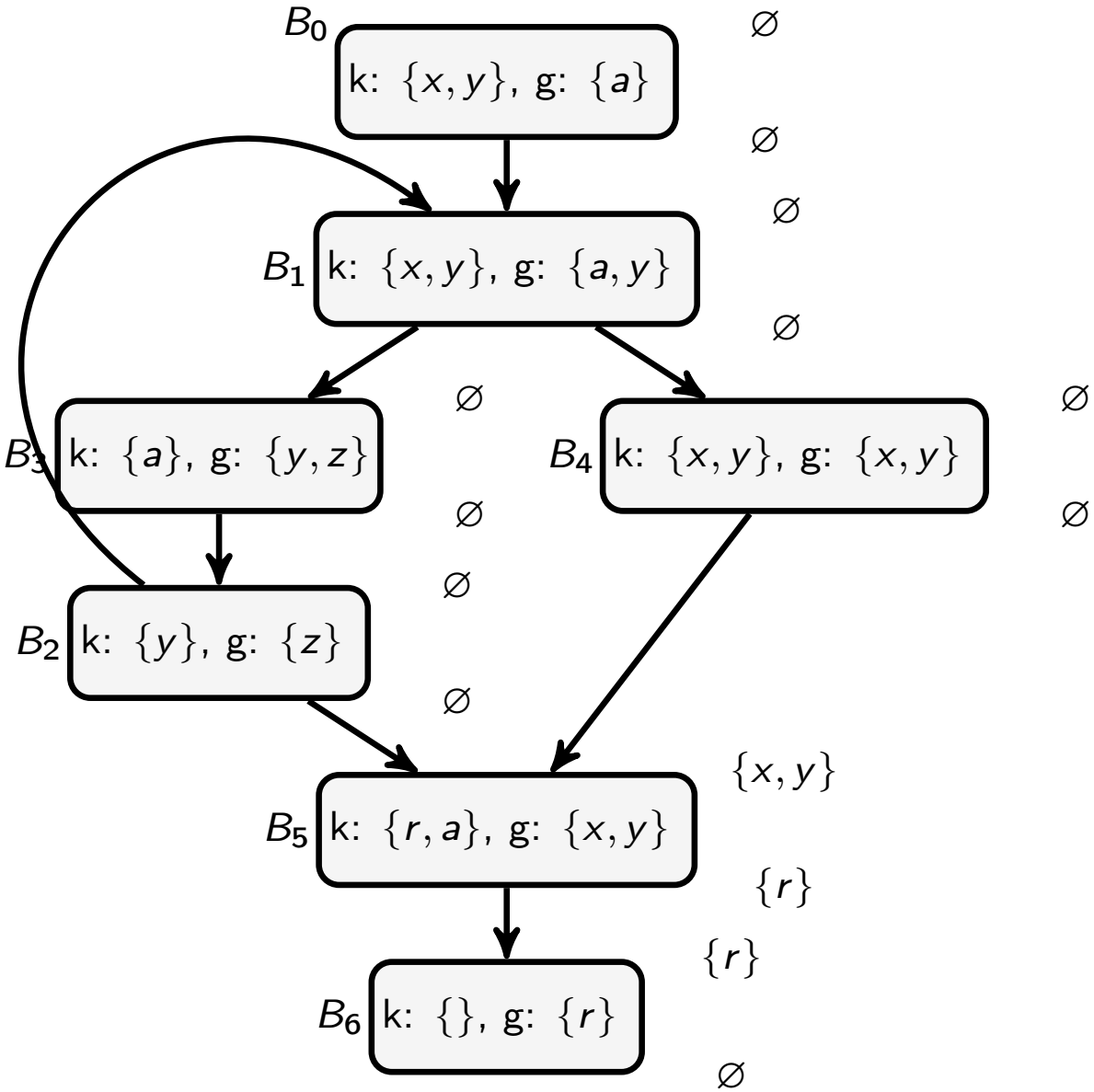
Example once again



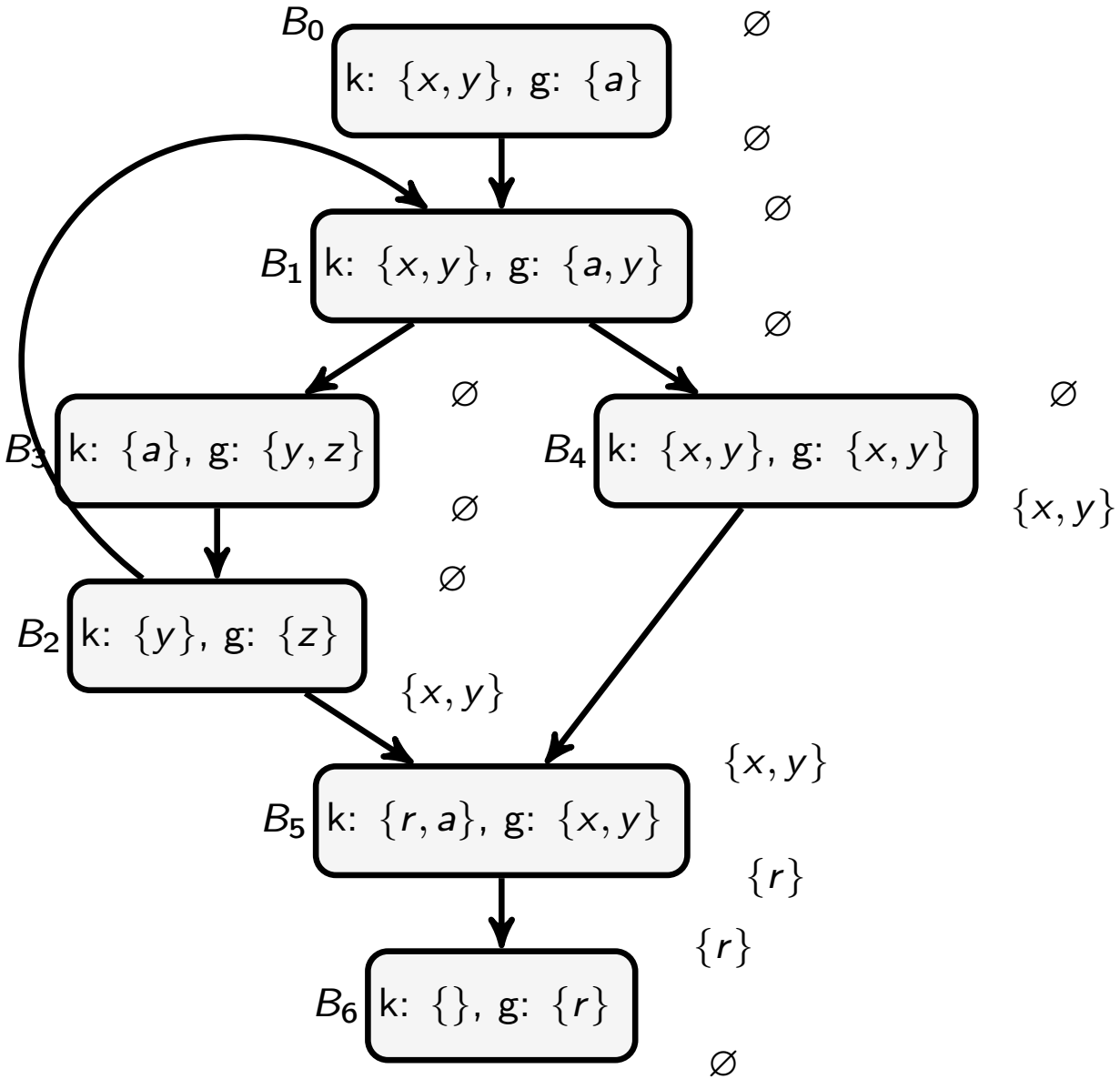
Example once again



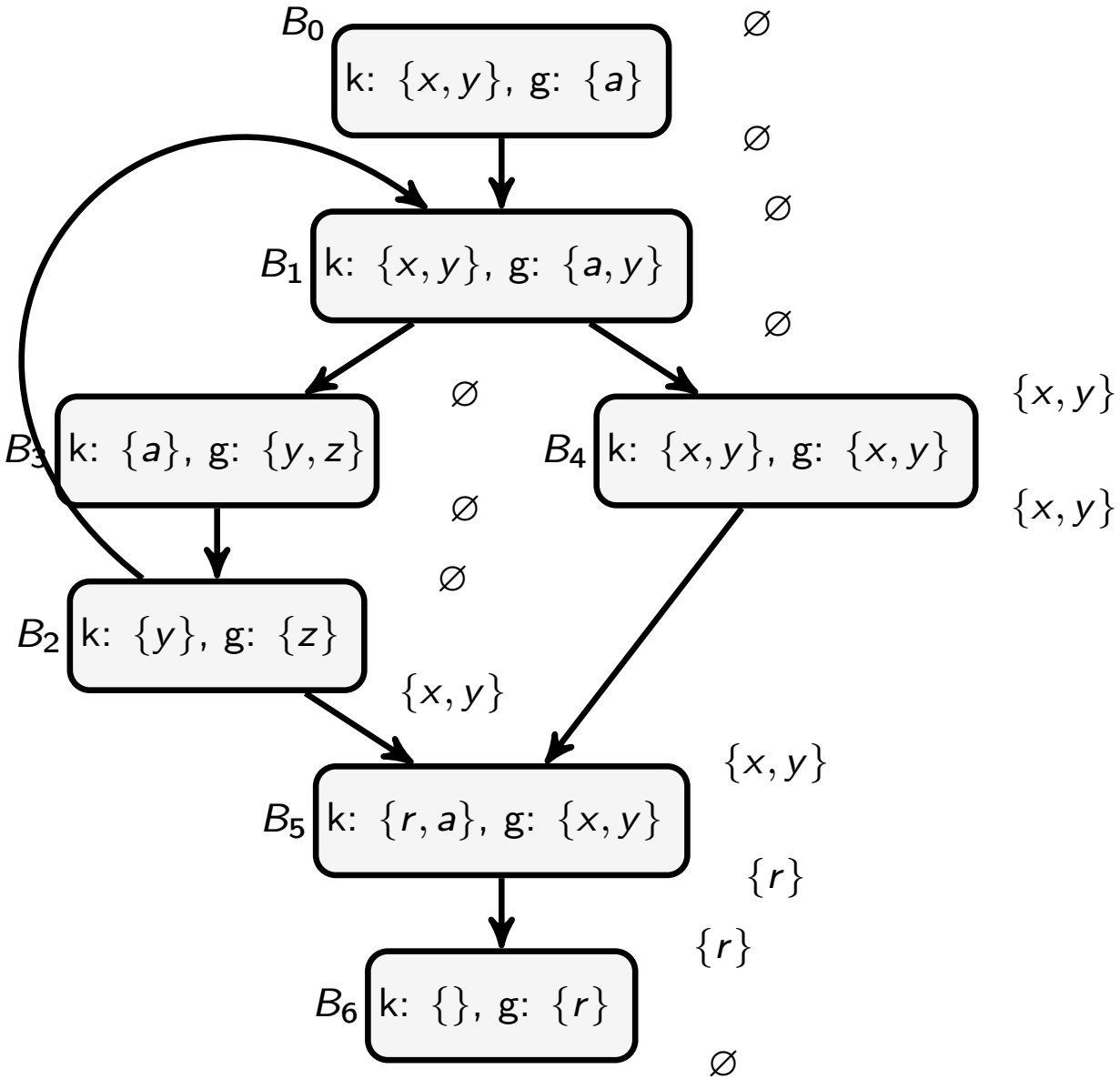
Example once again



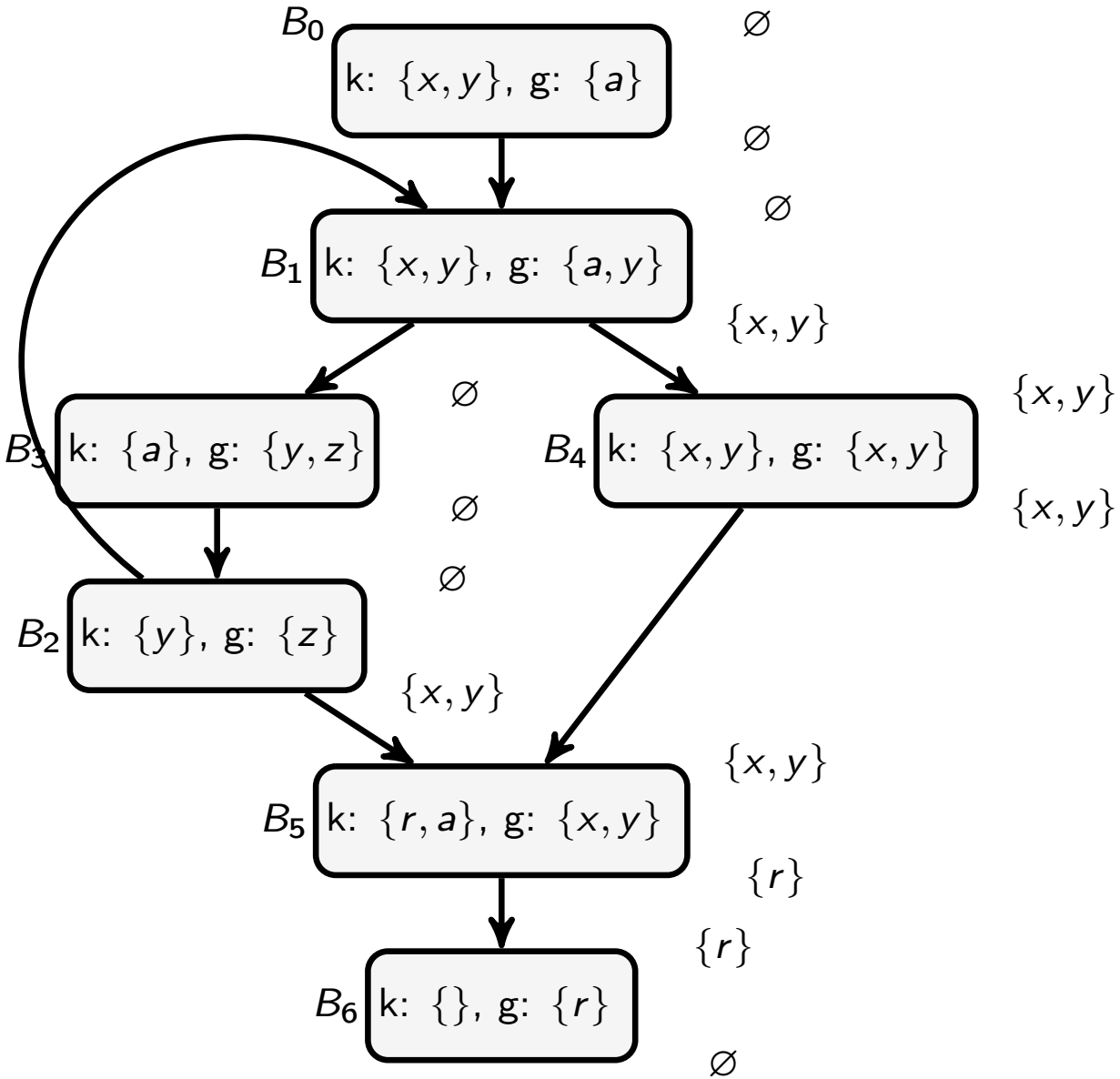
Example once again



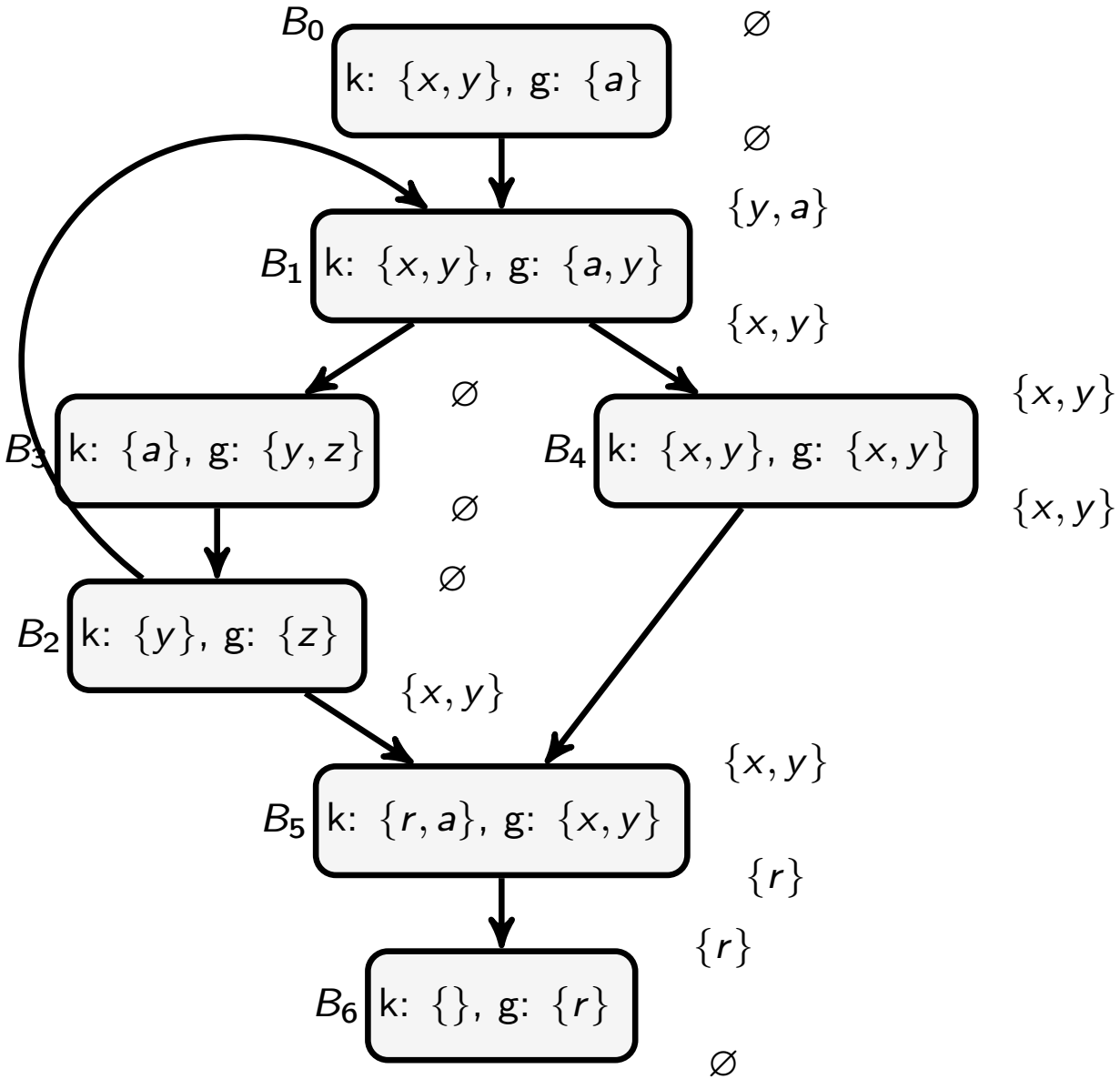
Example once again



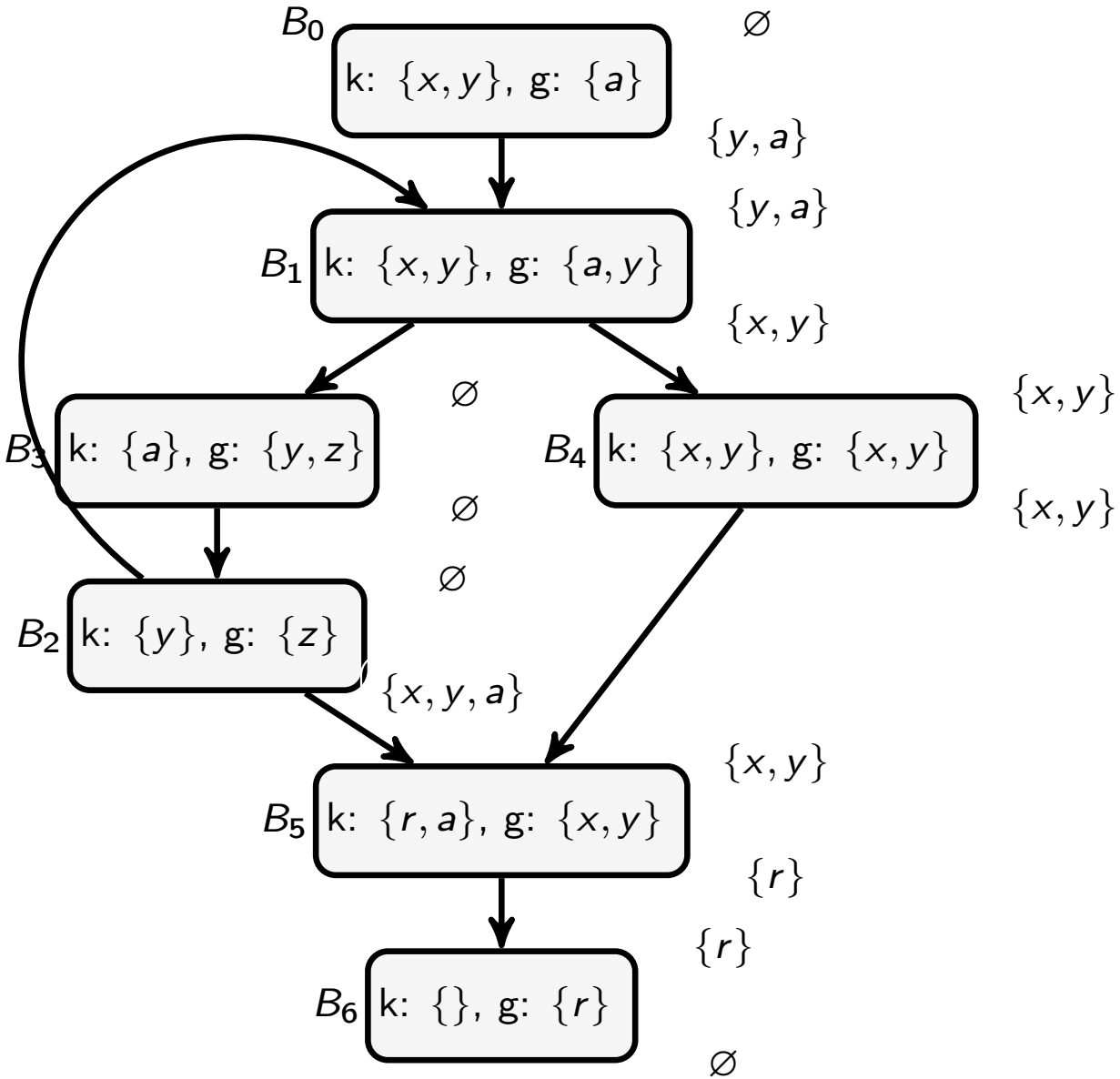
Example once again



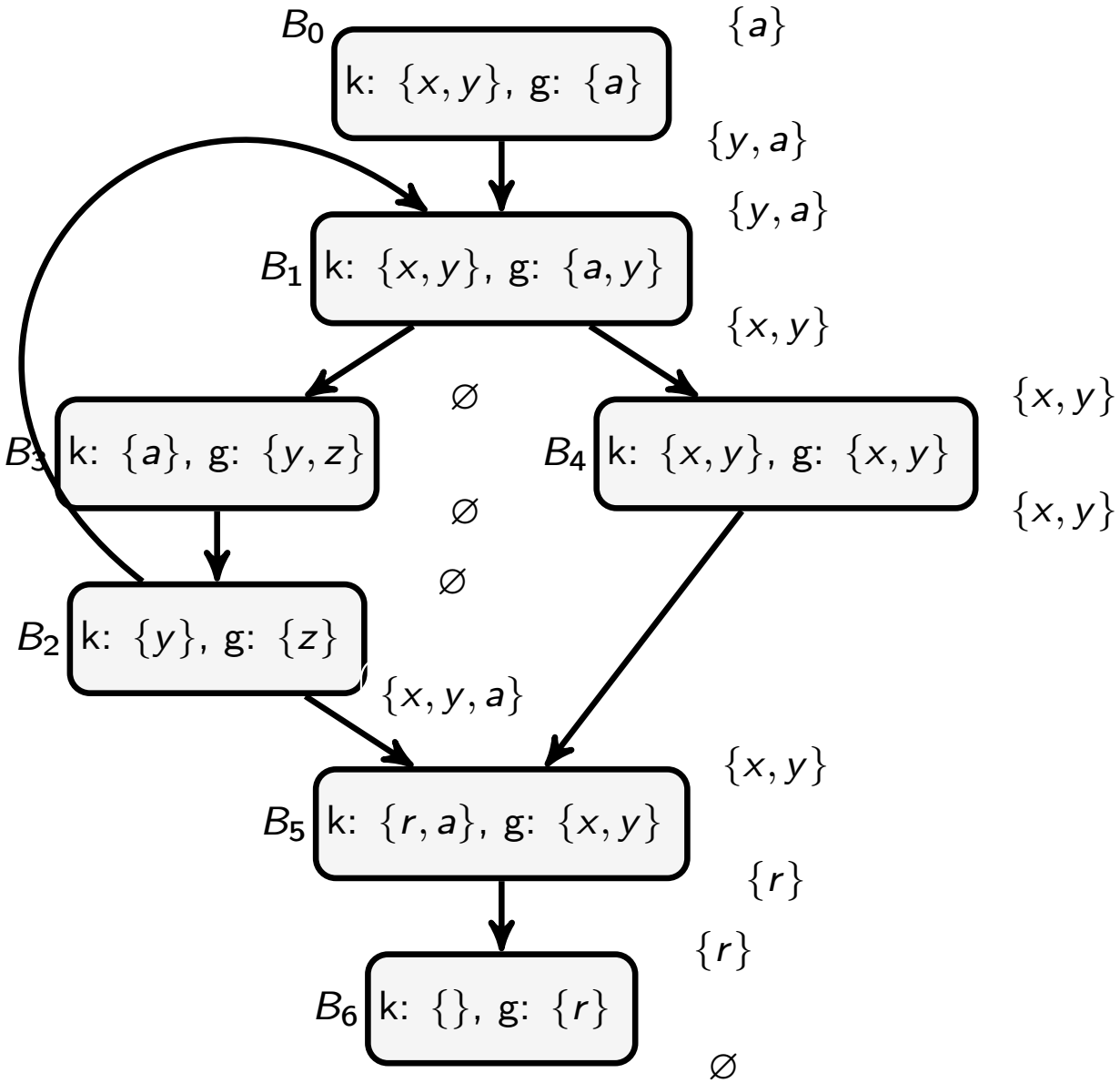
Example once again



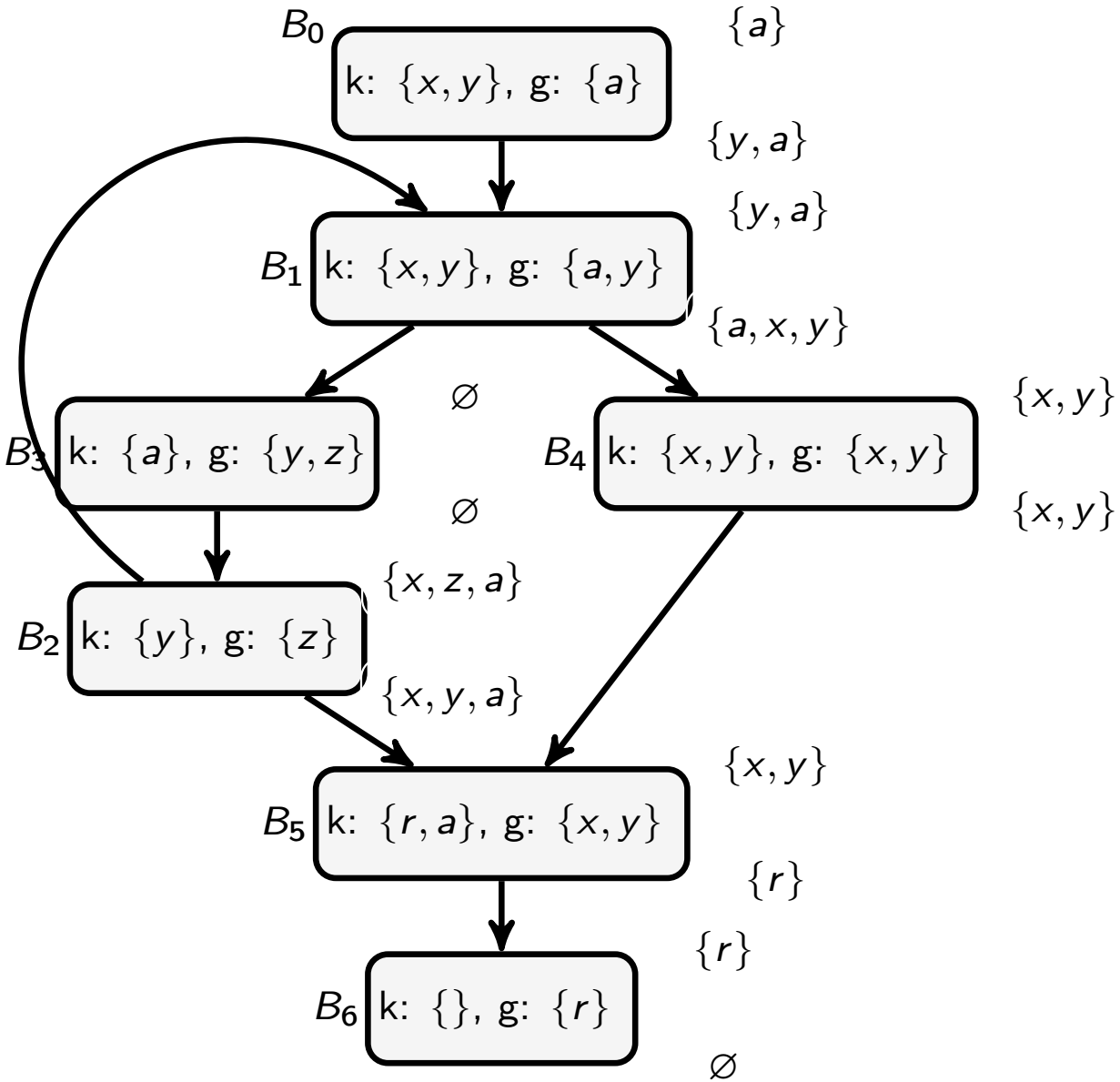
Example once again



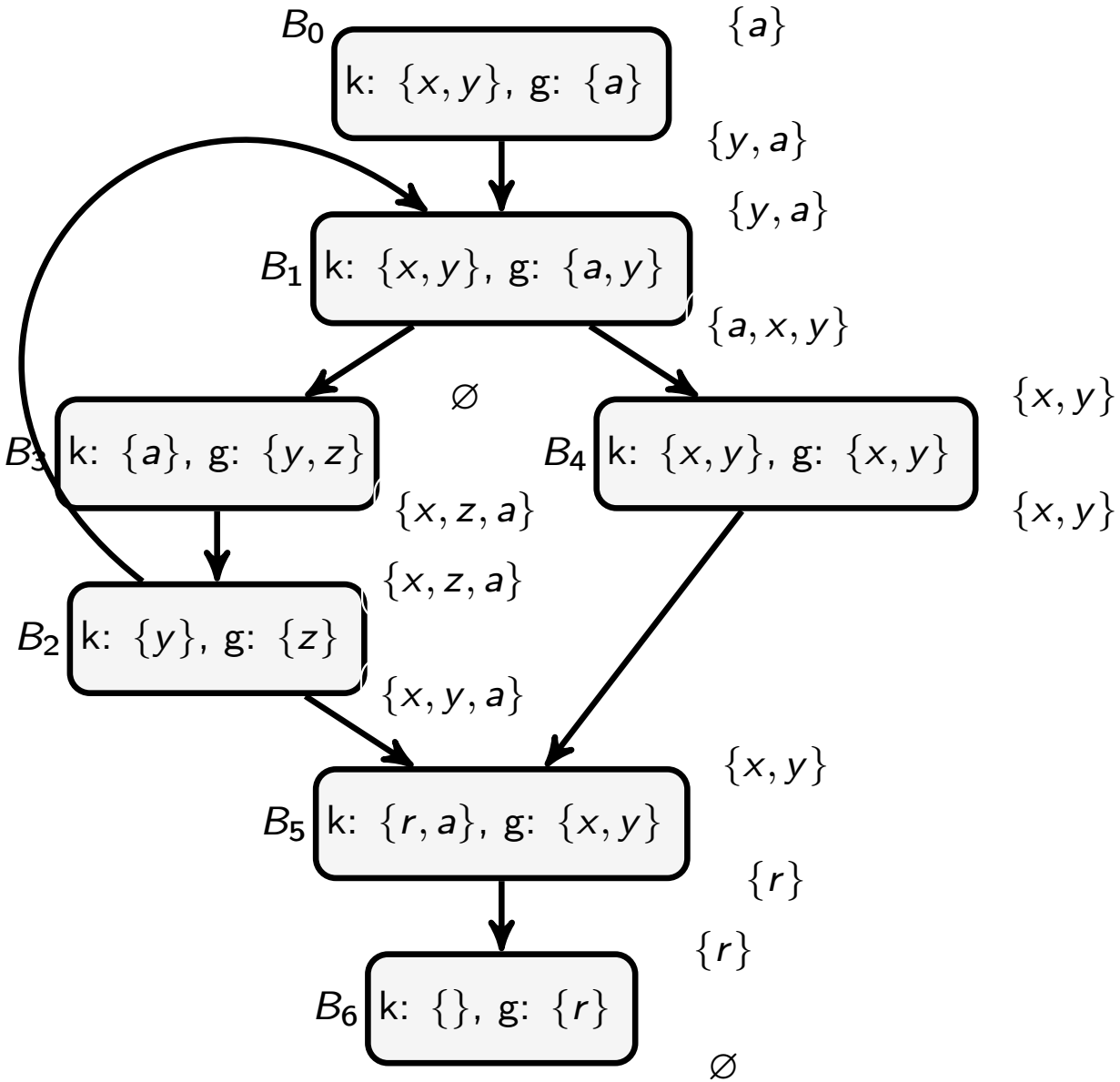
Example once again



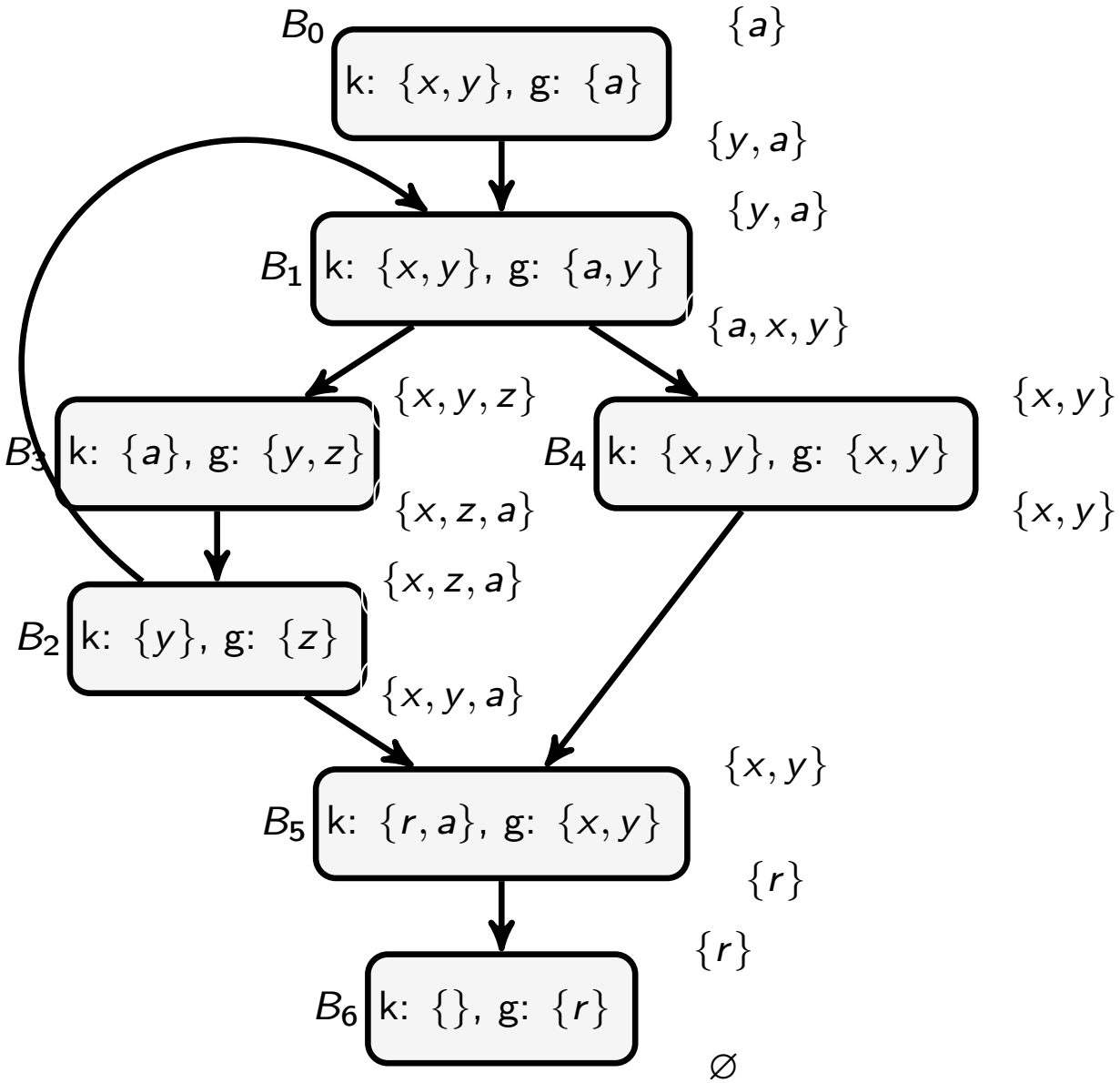
Example once again



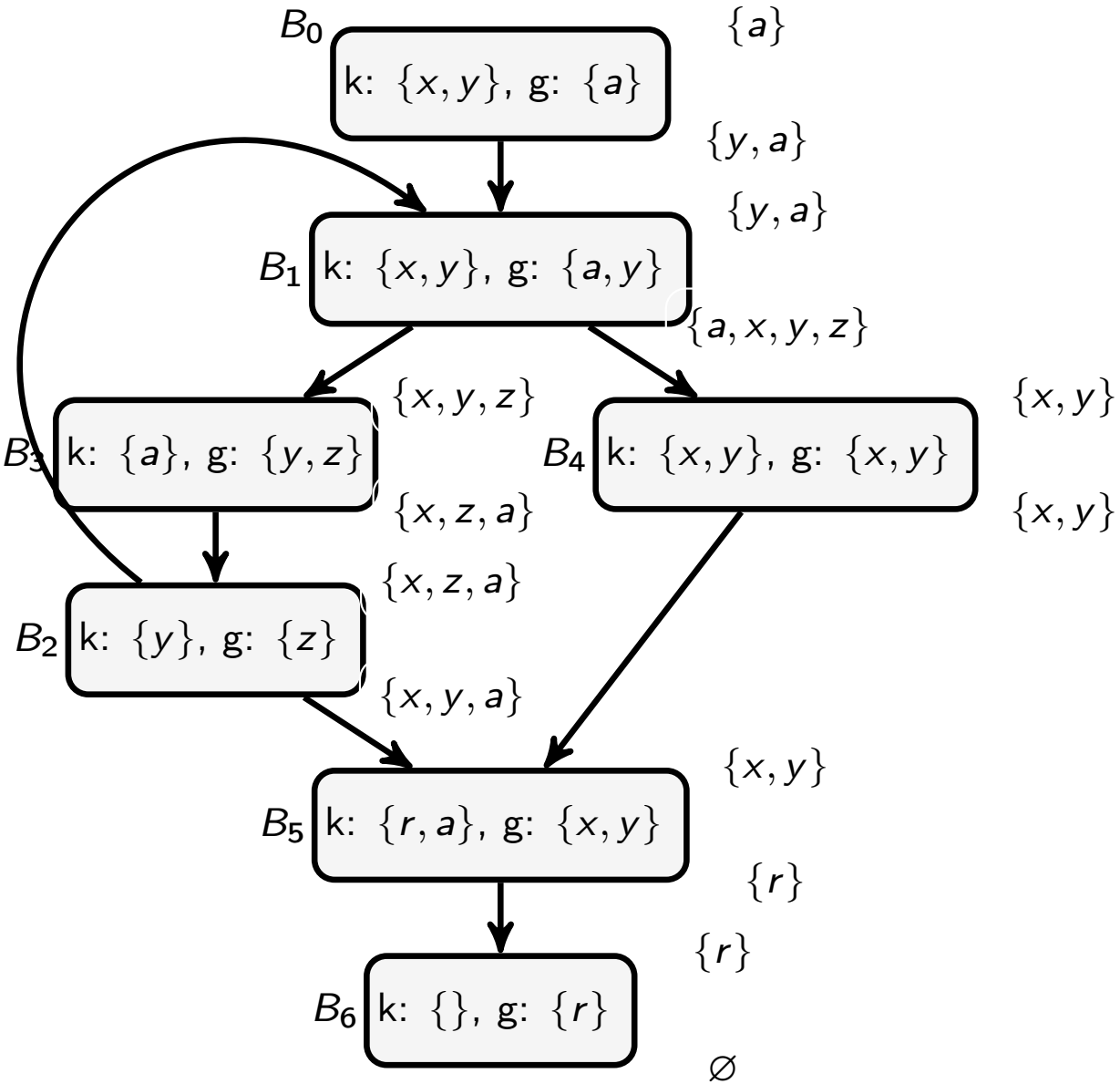
Example once again



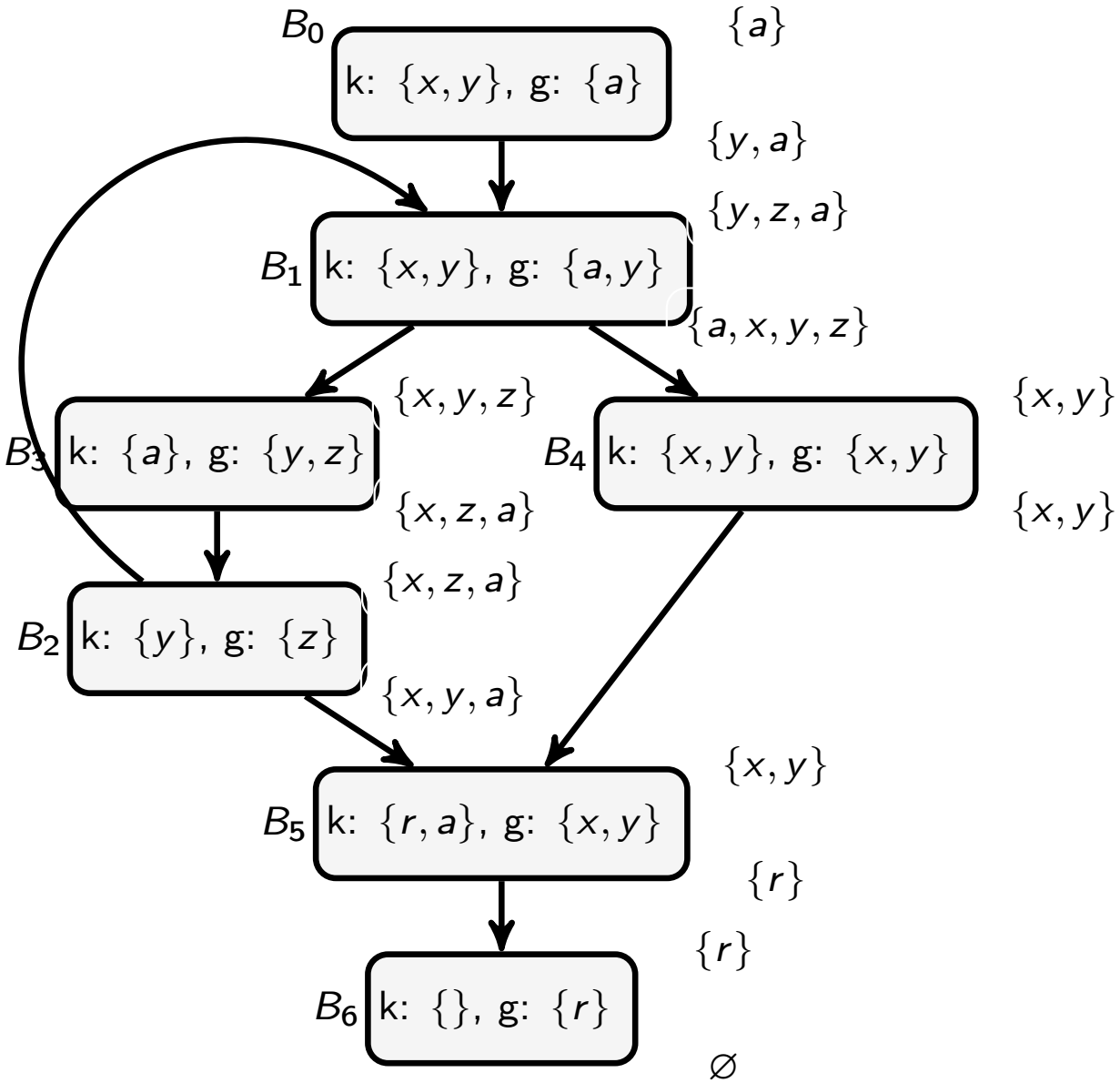
Example once again



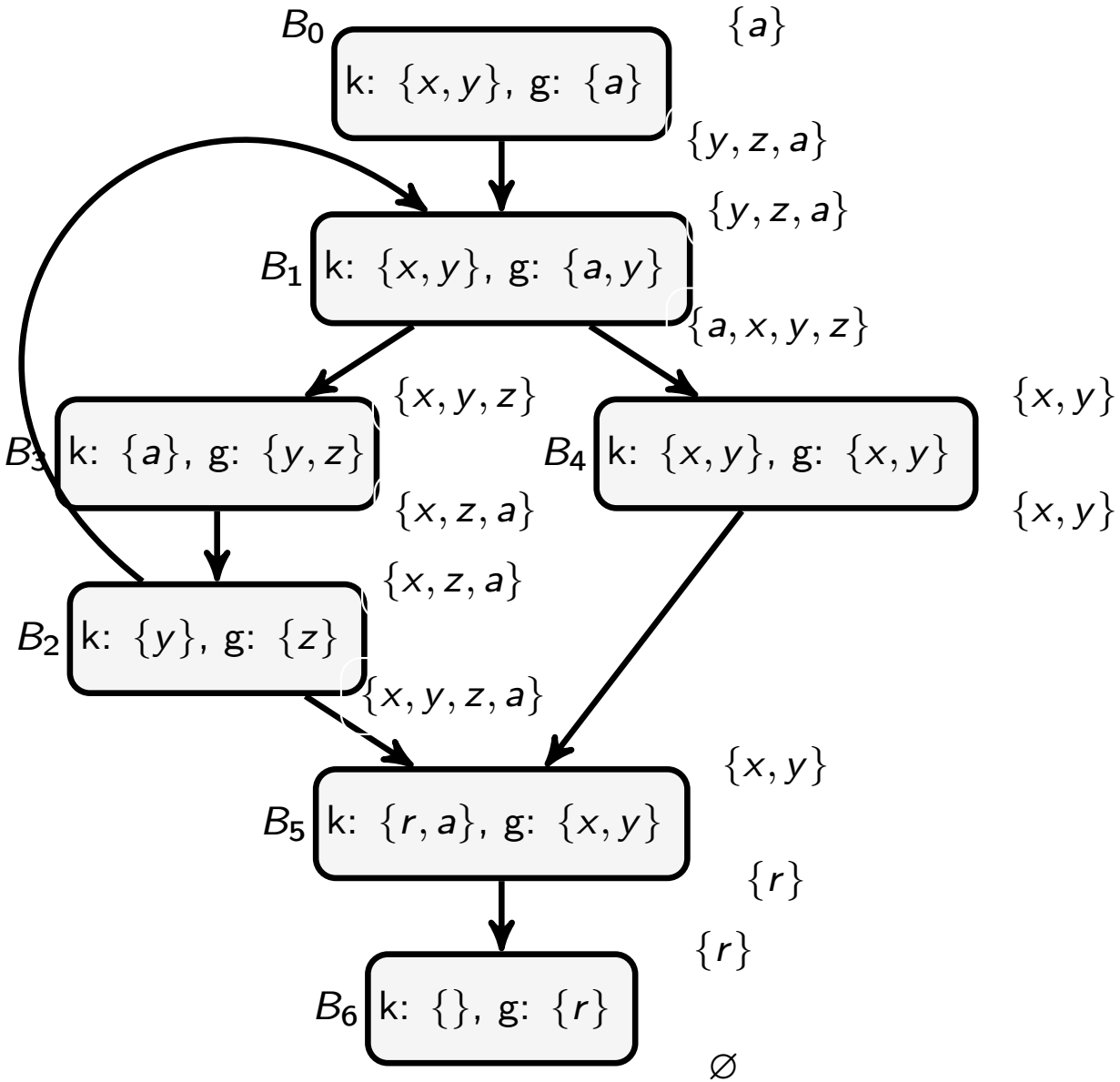
Example once again



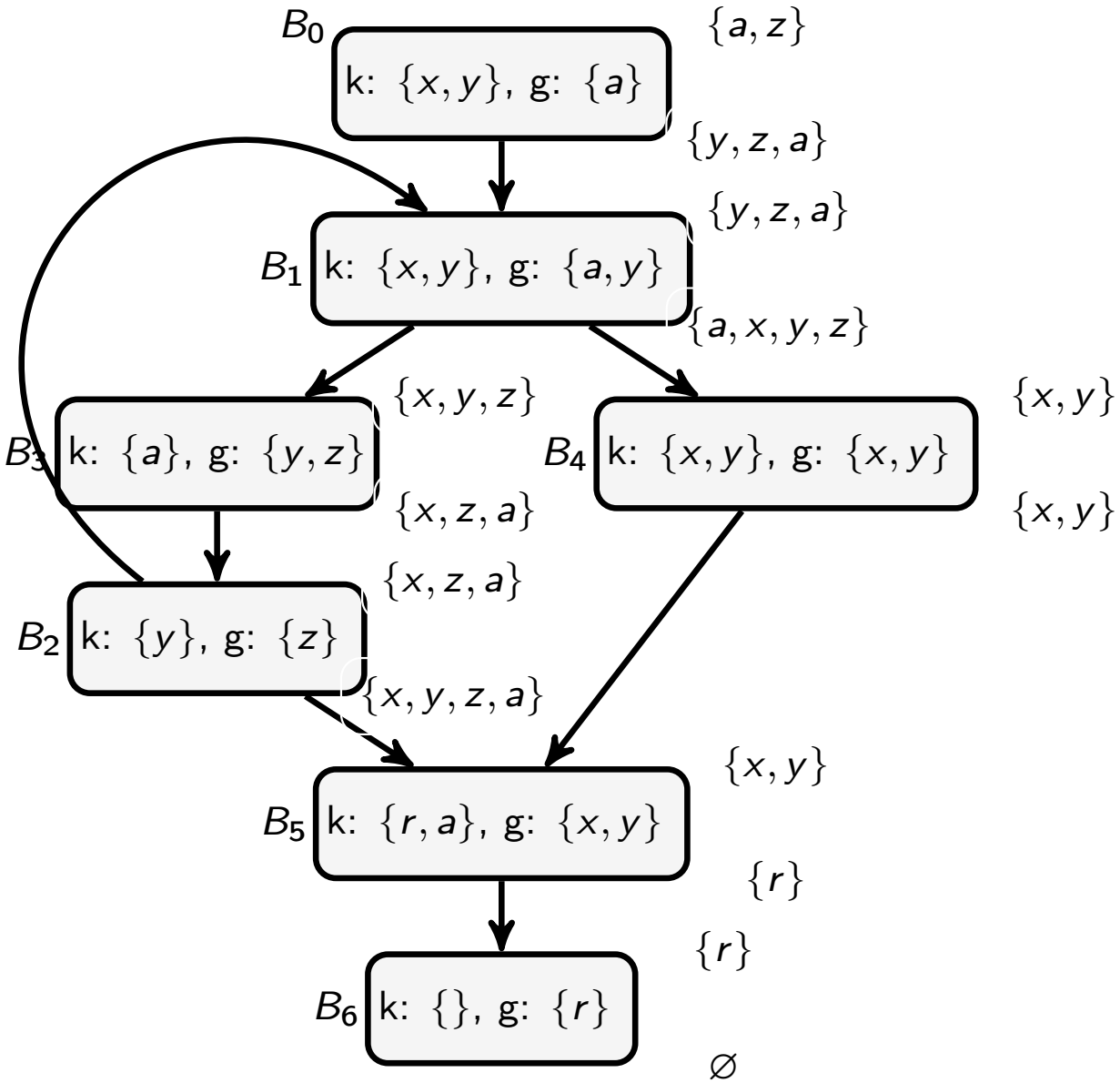
Example once again



Example once again



Example once again



Outline

References I

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007).
Compilers: Principles, Techniques and Tools.
Pearson,Addison-Wesley, second edition.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986).
Compilers: Principles, Techniques and Tools.
Addison-Wesley.