Universitetet i Oslo
Institutt for Informatikk
Reliable Systems
Martin Steffen, Gianluca Turin

INF 5110: Compiler construction

Spring 2021

Series 8

11. 05. 2021

Topic: (Intermediate) code generation (collection of exam questions)

Issued: 11. 05. 2021

Most of the exercises here are from earlier (written) exams. Some quite some years ago. They are also incorporated in the exam collection.

The first exercise here is *not* from an exam, but it's more of a recap anyway. The remainder are from the exams 2007, 2009, 2010, 2011, 2013, 2016. There might be even more years with questions about intermediate code generation or code generation. No need to include them all in this sheet, one can look at them in the exam collections.

Except 2016, the exams were not formulated by me. I somehow got hold of the exams, partly via students who had copies, and by previous lecturers. The older exams were mostly in Norwegian, so the text for those qis *not* the official exam text (but the meaning is preserved). I sometimes don't have access to official solutions in those years (of the exams resp. the corresponding exercises). Neither do I know how it was graded, for instance to which degree alternative formulations gave points, nor did I myself try to find all possible ways one could halfway correctly give an answer.

Actually, in some cases, I don't give a solution at all, because I have not solved them yet. Not because they are too complicated, simply because I have not found the time to work out solutions to all past generations of exams.

The purpose of the exercises (and going through old exams) is anyay not that I solve them, but that the participants try their knowledge and gain experience by doing them themselves.

Exercise 1 (Code generation) In this exercise we look at the code generation from the *notat* (i.e., from (Aho et al., 1986, page 538. . .))

(a) This is mean as repetition from the lecture. In the section for code generation, there is an example for which we showed at the very end of the section the resulting machine code. Look at the details how this algo generates this sequence. Try to determine a code sequence which is better (but does the same) than the one from that example. For the code, see the slide with the title *Code sequence* at the end.

(b) Discuss possibilities how one could improve the given algorithm from the lecture (taken from that book/notat).

(c) Translate the TAIC from Listing 1 to machine code using the algo from the notat/lecture. Consider some variations and improvements discussed in the previous point.

Assume that

- there are two registers initially "empty" and
- assume that for division "/", both source and destination have to reside in registers.

Listing 1: 3AIC

```
1 t := a − c
2 u := a + c
3 w := a / t
4 d := w + u
```

Solution:

(a) The task here is just to read and think through the corresponding material. Cf. the notat, resp. the slides of the corresponding chapter with headers like "Code generation algo for `x := y op z`" and the following, also the material/slides about `getreg`-function).

(b) The answer to this question is (partly) implicitly given by remarks in the lecture and the script in which way the code generation is "simple" and restricted.

One remark was that the code generation is unaware of the "semantics". One simple example is that some operations are symmetric (addition, multiplication). So, semantically and as far as the result is concerned, $x := y$ **op** z and $x := z$ **op** y are the same. It might, of course, happen that, for example the values of y and/or z are contained in the registers in such a way that one of the two (otherwise equivalent) variants is preferable (less "cost" in the given cost model). There are other such improvements (like for instance, the addition in $x := y + 0$ needs not to be performed).

These are *simple* examples that can be understood as transformations on the "source code" (which in this case means 3AIC). These improvements are simple in that they concern one line only.

Possibilities of improving the code generation taking into account more than one such line are basically limitless (depending on how much "intelligence" and "semantic analysis" and computational effort one is willing to invest. An improvement could for instance try to *swap 2 neighboring lines* (or more) in case the semantics is unaffected by that (no data dependencies) an see if it leads to an improvement.

Attempting a truly optimal code (even restricted to basic blocks), while theoretically thinkable, is typically not attempted.

In the lecture we have hinted at the *liveness* information gives a good handle on an improvement. Already the simplest form, taking into account one single basic block, is an important improvement (and the algo sketched in the code generation chapter takes that into account). As mentioned, one could make the liveness information also "global" spanning more than one basic block. One might debate, whether that should count as an improvement of code generation or rather basically the same code generation, only relying on better liveness information. The code generation algo, though, would need an adaptation (and improvement thereby), as at the end of the block, registers need not be "flushed back" unconditionally.

Liveness analysis is important, but there are other "semantical properties" which one could analyze (for instance to avoid re-computation).

If one moves one value from the memory into a register, and the op destroys that due to the specific form of the code generation here (for instance as done by the first instruction in our example), then it might be worthwhile to copy the value into a second register (to keep it for further use). That would rely on liveness information, as well. One may also take into account, how "long" in the future the value will be needed again. If the next use is in the very near future, such a copy should lead to an improvement, if far into the future, it may not (the register way well be purged until the next use, and the copy was for nothing, resp. the copy costed time in the cost model for no other gain.)

(c) The code is given in Listing 2. For the alternative, where the 3AIC replaces $d := w + u$ by $d := u + w$: We only have to look at the last 2 lines, as the previous lines are unaffected. In first approximation, the code generation works line by line. The code generated by one line is influenced by the code generated from the previously lines in that it takes into account the current status of the registers. However, the code generated "in the future" does not influence the code generated for one line of 3AC. On the other hand, it's not strictly true that the "future source code" has no influence on the code generation.

Listing 2: Generated code

```
1  MOV a R0
2  SUB b R0    // t in R0
3
4  MOV a R1    // what a pity: reload a
5  ADD c R1    // u in R1
6              // both regs full, one of
7              // them needs to be ``purged''
8              // We choose R1 (containing u)
9              // as t (in R0) will soon be
10             // used:
11 MOV R1 u    // save value for u ``back home''
12
13 MOV a  R1    // t still in R0
14 DIV R0 R1    // w in R1
15              // R0 is ``free'' as t is no
16              // longer needed (not live)
17
18 ADD u R1     // w is lies perfectly in R1 already
19 MOV R1 d     // copy d's value to home position
```

Listing 3: Generated code, changed last 3AIC line

```
1  MOV a R0
2  SUB b R0    // t in R0
3
4  MOV a R1    // what a pity: reload a
5  ADD c R1    // u in R1
6              // both regs full, one of
7              // them needs to be ``purged''
8              // We choose R1 (containing u)
9              // as t (in R0) will soon be
10             // used:
11 MOV R1 u    // save value for u ``back home''
12
13 MOV a  R1    // t still in R0
14 DIV R0 R1    // w in R1
15              // R0 is ``free'' as t is no
16              // longer needed (not live)
17 ---------- below here: alternative code d := u + w ----------------------
18 MOV u R0
19 ADD R0 R1
20 MOV R1 d
```

□

Exercise 2 (Code generation (-%))

(a) Given is the program from Listing 4. The code is basically *three-address code,* except that we also allow ourselves in the code *two-armed* conditionals and a while-construct (with the conventional meaning). The input and output instructions in the first two lines resp. the last two lines are considered as standard three-address instructions, with the obvious meaning of "inputting" a value into the mentioned variable resp. "outputting" its value. We assume that *no* variable is live at the end of the code.

Listing 4: 3-address code example

```
1  a := input
2  b := input
3  d := a + b
4  c := a * b    // <- looky here
5  if ( b < 5) {
6    while (b < 0 ) {
7      a := b + 2
8      b := b + 1
```

```
 9      }
10      d := 2 * b
11  } else {
12      d := b * 3
13      a := d - b
14  }
15  output a
16  output b
```

Which variables are *live* immediately at the end of line 4. Give a short explanation of your answer.

Solution: One way to answer that problem is to draw the control-flow graph (just for the overview) and go through the steps of the liveness algo. But actually, the program is simple enough so one might even more easily just look at the program and figure out by "carefully thinking" which of the variables at the specific line are live and which are not. Note: it's *not* required to give the values for the *inLive* and *outLive* points throughout the CFG. Other exam questions *do* require the full construction (partition the intermediate code, show the CFG, and show the liveness result for all positions in the graph), but here one is allowed to simply give the result (it's easy enough).

But even more central is, to simply list the variables for which the info is needed (a, b, c, d). Since the task does not require to formally use the algorithm to derive the answer or even give the CFG, we simply give the liveness status straight:

a: That's a tricky one. But it's live! In the else-branch, the first thing to happen to a is that it's assigned to ("defined"). So in that branch, it is dead. In the true-branch, it's assigned to also, but it's inside the while-loop. If it so happens that the while-loop *is not executed at all*, then obviously the assignment to a will not happen. Which means, the first thing to happen to a is the output-statement in line 15. That most definitely counts as "use" of a. It is important to realize that `it does not matter` whether the while-loop actually is executed or not (we are technically dealing with *static* liveness). We are conceptually operating on the CFG, where there are 2 possiblities: the while-loop is entered, or not. Since statically we don't know what *actually* happens, we have to take both options into account. Therefore, as said, a is live.

b: The variable is immediately live as it is used in the next line.

c: There variable is never "used". It's only mentioned in live 4, where it's assigned to ("defined") but afterwards never even mentioned (and not before either). So, being a "write-only" variable, it's completely useless, and more specifically dead after line 4.

d: This variable is more interesting again. Like b, it's assigned to in both branches of the conditional, but unlike b, it's not assigned-to (in the false-branch) inside the while-loop. So, unavoidably, in both cases, d is overwritten before it's used again in the output statement in line 16. Therefore, d is dead.

Exercise 3 (Code generation (%))

Consider the following program in 3-address intermediate code.

Listing 5: 3-address code example

```
1  a := input
2  b := input
3  t1 := a + b   // line 3
4  t2 := a * 2
5  c := t1 +  t2
6  if a < c goto 8
```

```
7   t2 := a + b
8   b := 25       // line 8
9   c := b + c
10  d := a − b
11  if t2 = 0 goto 17
12  d := a + b
13  t1 := b − c
14  c := d − t1
15  if c < d goto 3
16  c := a + b
17  output c      // line 17
18  output d
```

(a) Indicate where new *basic blocks start*. For each basic block, give the line number such that the instruction in the line is the first one of that block.

(b) Give names B_1, B_2, . . . for the program's basic blocks in the order the blocks appear in the given listing. Draw the *control flow graph* making use of those names. Don't put in the code into the nodes of the flow graph, the labels B_i are good enough.

(c) The developer who is responsible for generating the intermediate TA-code assures that temporary variables in the generated code are *dead* at the end of each basic block as well as dead at the beginning of the program, even if the same temporary variable may well be used in different basic blocks.

Formulate a general rule to *check* locally in a basic block whether or not the above claim is honored or violated in a given program.

Assume that all variables are dead after the last instruction.

(d) Use the rule formulated in the previous sub-problem on the TA-code given, to check if the condition is met or not. The temporary variables are called t_1, t_2 etc. in the code.

(e) Draw the control flow graph of the problem and find the values for *inLive* and *outLive* for each basic block. Consider the temporaries as ordinary variables.

Point out how one can answer the previous Question 4.d directly after having solved the current sub-problem.

Are there instructions which can be omitted (thus optmizing the code)? Are there variables which are potentially uninitialized the first time they are used.

Solution:

(a) The basic blocks are indicated as comments in the code. The line numbers shift therefore, of course.[1] The first line indicates a basic block, targets of (conditional) jumps indicated basica blocks, and lines after (conditional) jumps indicate basic blocks.

Listing 6: 3-address code example: basic blocks added

```
1   // ————————— B1 —————————
2   a := input
3   b := input
4   // ————————— B2 —————————
5   t1 := a + b   // line 3
6   t2 := a * 2
7   c := t1 +  t2
8   if a < c goto 8
9   ————————— B3 —————————
10  t2 := a + b
```

[1]Note that many representations, for instance in our lecture, favor 3AIC, where one uses *symbolic* labels not actual line numbers. That's a better way of dealing with the issue of (conditional) jumps in intermediate code, anyway. The same applies to assembly code.

```
11 ─────────── B4 ──────────────
12 b := 25        // line 8
13 c := b + c
14 d := a − b
15 if t2 = 0 goto 17
16 ──────────── B5 ─────────────
17 d := a + b
18 t1 := b − c
19 c := d − t1
20 if c < d goto 3
21 ──────────── B6 ─────────────
22 c := a + b
23 ──────────── B7 ─────────────
24 output c      // line 17
25 output d
26 ─────────────────────────────
```

(b) For the CFG. see below in e)

(c) A possible rule could be

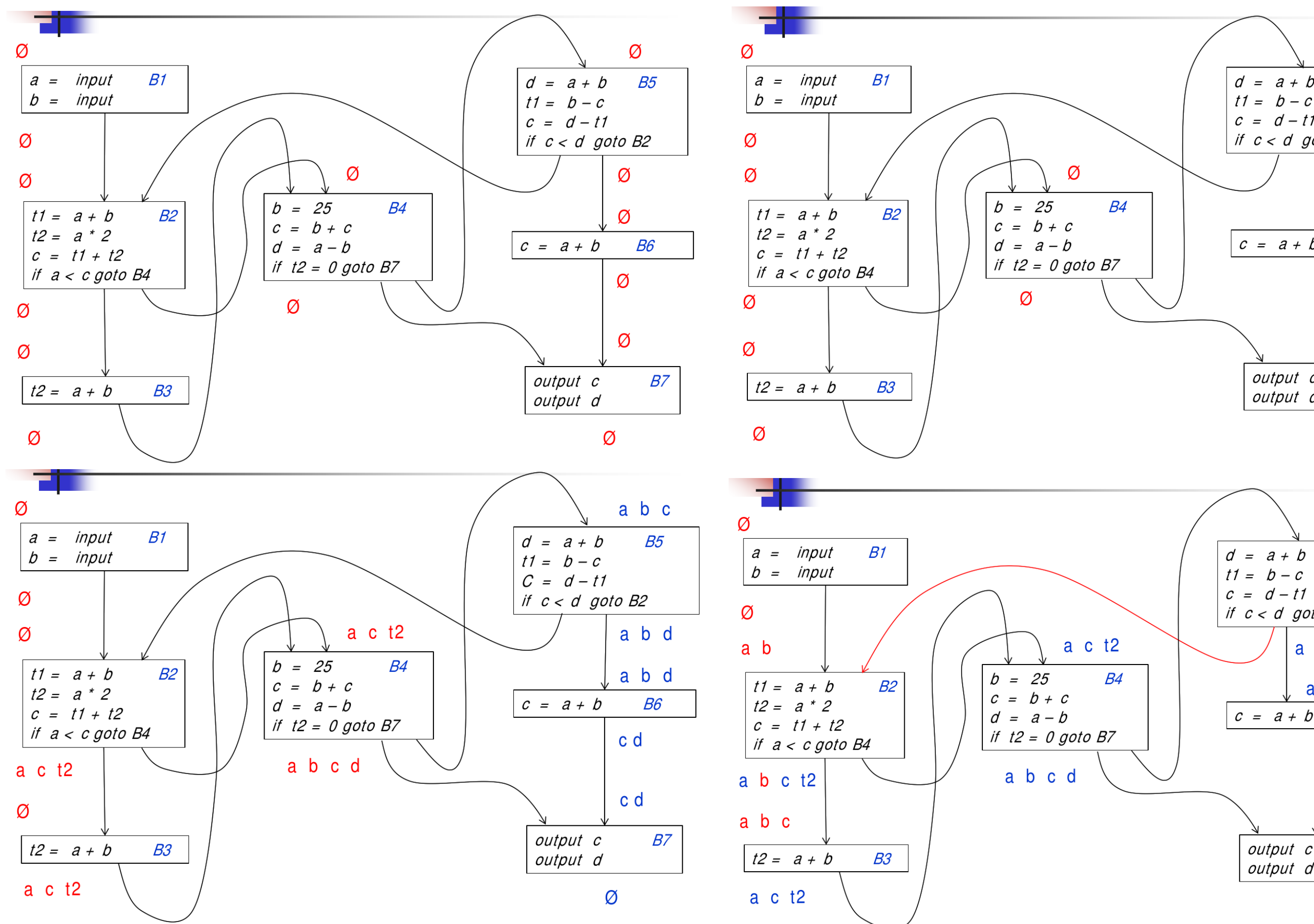
> All temporaries which are *used* in a given basic block must be assigned to ("defined") in the same before the (first) use.

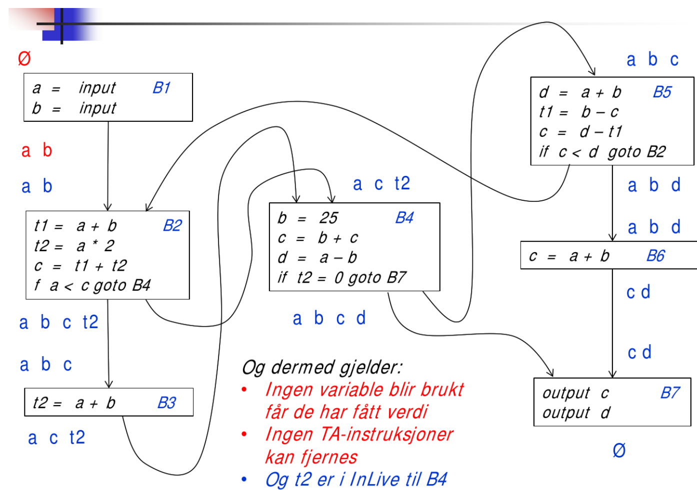
Another way of saying it is:

> No temporary variable must have a "next-use" at the beginning of a basic block.

(d) sanitary check: In block B_4, the temporary t_2 violates the formulated rule.

(e) Liveness:





Exercise 4 (Code generation (–%))

(a) Arne has looked into the code generation algo at the end of the notat (from (Aho et al., 1986, Chapter 9)). He surmises that for the following 3AIC

```
1  t1 := a − b
2  t2 := b − c
```

the code generation algorithm will produce the machine instructions below. He assumes two registers, both empty at the start.

Listing 7: 2AC

```
1  MOV   a, R0
2  MOV   b, R1
3  SUB  R1, R0
4  SUB   c, R1
```

Ellen disagrees. Who is right? Explain your answer.

Solution: Arne is wrong. The code is not as it is generated. The code as such makes "semantical" sense, it's just not code that is being generated according to the code generation from Aho et al. (1986). How can we easily see that? What makes the code generation a bit weird is that the *machine* code is a two-address code and that it uses the two operands in some peculiar way, in particular, it determines first a location where the result should go. The preference is *strongly* that the result is supposed to end up in a register. Even if the registers are all "full" still the code will put the result in a register (but of course saving the content back to main memory). The circumstances when or how that happens are not fully given in the book. However, as long as there are *free* registers, a register is taken for the result. The second step is: is the *first operand* (by happenstance) already in that register. Well, as the exercise states: we have 2 registers, both are empty. Therefore 1) the result will end up in a register, say R_0, and 2), we have to move the first operand into that register. So the first line of the code is still fine. It's the second line where the shown code deviates from the presented code generator: The "second" step is *always* the execution of the operation itself (of course, if the first step is missing, the "second" step is actually the first).

So: an *easy* way to see that the code generation won't generate the code of Listing 7 is: the code generator always translates the prototypical 3AIC assignment with a binary operator (the one we discussed in the lecture)

into 1 or to 2AC assinments: either just "`OP...`" or `MOV` followed by "`OP`".

Therefore, independent from whether the above sequence makes semantically sense or not: the code generator won't generate it.

It's not part of the question, but here's the code which would be generated

Listing 8: 2AC (not part of required answer)

```
1                // t1 is not in a register, so we choose one (R0) and then
2  MOV a, R0     // load first operand to that register.
3                // This register is also which contains the result
4  SUB b, R0     // do the substraction.
5  MOV b, R1     // the second line is translated analogously.
6  SUB c, R1     // a is not live after the first 3AIC code, we could
7                // reuse R0 therefore!
```

□

Exercise 5 (Code generation & P-code (25%))

`lda v`	"load address"	Determine the address of variable v and push it on top of the stack. An address is an integer number, as well.
`ldv v`	"load value"	Fetch the value of variable v and push it on top of the stack
`ldc k`	"load constant"	Push the constant value k on top of the stack
`add`	"addition"	calculate the sum of the stack's top two elements, remove ("pop") both from the stack and push the result onto the top of the stack.
`sto`	"store"	
`jmp L`	"jump"	goto the designated label
`jge L`	"jump on greater-or-equal"	similar conditional jumps ("greater-than", "less-than" ...) exist.
`lab L`	"label"	label to be used as targets for (conditional) jumps.

Table 1: P-code instructions

(a) This sub-task is to design a *"verifier"* for programs in P-code, i.e., for sequences of P-code instructions.

  (i) List a many possible "properties" that the verifier can or should check or test in P-code programs. Explain in which sense a P-code program is correct given the list of properties being checked for.

  (ii) Sketch which *data structures*

(b)

(c) We want to translate the P-code to machine code for a platform where all operations, including comparisons, must be done between values which reside in *registers* and that register-memory transfers must be done with dedicated `LOAD` and `STORE` operations. During the *translation,* we have a *stack* of descriptors.

Consider the P-instruction

`ldv b`

where b is a variable whose value resides in the home position. This instruction therefore pushes the value of b onto the top of the stack. When translating that to machine code, a

question there is what is better: 1) doing a `LOAD` instruction so that the value of b ends up in register or alternatively 2) push a descriptor onto the stack marking that b resides in its home position.

Discuss the two alternatives under different assumptions and side conditions. These may include the whether the user-level source language assures an *order* of evaluation of compound expressions. Other factors you think relevant can be discussed as well.

(d) Again we translate our P-code to machine code and, as in the previous sub-problem, we assume we translate again one block at a time, in isolation, and that consequently all registers have to be "emptied" at the end of a basic block in a controlled manner.

The question is to find out which *data descriptors* in the stack are needed and if other kinds of descriptors are needed.

We assume that we can *search* through all the descriptors of the elements on the stack each time this information is needed. In that way, we avoid having to add another layer of descriptor(s).

With your descriptor design: describe how to find information needed during code generation and, if your design contains additional descriptor, how to make use of them.

Solution:

(a) (i)

(ii)

(iii)

(b)

(c) The following is from the given solution at that time.

(i) If the language definition specifies that the evaluation order is fixed from left-to-right, one should generate a `LOAD` instruction to get the value into the registers. If the language definition leaves the order open, it may be better *not* to load the variable but a corresponding descriptor into the stack. Remember that the stack is *not* a run-time stack, it's a data structure the code generator uses to perform it's task. Insofar that the code generator goes through the intermediate code (here P-code) of the basic block instruction by instruction, it does some form of "static simulation" of the P-code execution, including doing a form of simulation of the stack (in the simulation however, operating with descriptors). In that sense, it's a kind of "simulation" of a stack at run-time, but it's not what we call the stack of ARs of a typical, stack-allocated run-time environment.

(ii) The situation leaves room for many optimizations. One situation discussedq is that if the expression contains a function call (or method call etc). I would not cover that in this tasks, since I would not really consider that the expression then is part of *one basic block*. The call would lead to the situation that the basic block is split into (at least) two sub-blocks: before the call and after. It's not part of the lecture how the blocks and edges are done (i.e. how the CFG is done) in the presence of function calls. One proposed solution ignores that and treats a function call as being "inside" the basic block. The problem with function calls is that they can *change* values (the may have side effects). If there are side effects, the order of evaluation matters, if there are no side effects, the order does not matter. If therefore the expression is *side-effect free* there's no need to load the value directly, as it effectively does not matter when it's loaded. Therefore one may be better off simply using the descriptor stack marking where the variable is being found in memory.

(d) In any case we need the following

- if the argument is a constant (and which)
- if the value of the argument is a program variable (and which)
- if the value resides in a register (and in which)

Not everything possible will be recorded on the stack. Note that we don't record *on the stack* what is the content of the registers (only indirectly by saying whether or not a value can be found in this-and-that register).

It should be noted that the descriptors stack is not really good enough to keep track of all the information the code generator wants to keep an eye on. At least if it wants to keep a level of overview over registers and variables comparable to the code generator from the lecture. The reason why the stack itself is not good for that, no matter how much info we plan to store into the stack entries, is simply that popping arguments off the stack means, forgetting all information stored for the corresponding operand. The stack may easily become empty during the expression evaluation in the middle of a basic block, after which the code generator would not know where variables are etc.

Thus, one needs *additionally* to store such information, independent from the stack. Bascially, one would need, besides the stack, register descripters and address descriptors in the same way the code-generator from the lecture for 3AIC uses.

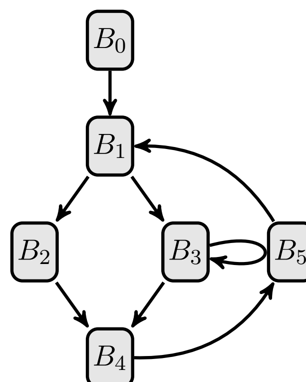
Exercise 6 (Code generation and analysis (25%))

(a) We partition a method in a program into *basic blocks* and draw the *flow graph* for the method. At the end we figure out which variable is *live* at the beginning and at the end of each basic block (for example useing the "iteration"-method). Answer the following questions:

(i) How can one find TA-instructions (if any) which are guaranteed *not* to have any influence when executing the program?

(ii) How can one determine whether there is a variable (optionally which ones) that are read ("used") before they have been given a value in the program?

(b) Take a look at the following control-flow graph



Knut opines that the graph contains the following *loops* (where loop is understood as defined in connection with code generation and control-flow graphs)

$$B_1, B_2, B_4, B_5$$
$$B_1, B_3, B_4, B_5$$
$$B_1, B_2, B_3, B_4, B_5$$

Astrid disagrees. Who is right? Give an explanation. If Astrid got it right, give the correct loops of the graph.

(c) The following TA-instructions are contained in block B_2 of the previous subproblem:

```
...
k = j + x
k = k * k
...
```

To save execution time, we wonder whether it is possible to move this code out of the smallest loop L what B_2 is part of. So:

(i) What do you have to check in the different basic blocks before you can do such a move safely, and in exact which blocks must such checks be done?

(ii) concretely: such an intended move will include that we add at one place outside L the following lines

```
...
k' = j + x    // k': new variable
k' = k' * k'
...
```

In addition, will we replace the original sequence (in B_2) with the assignment k = k'. Now: *where* outside loop L is it appropriate to move the (adapted) sequence to, which gives the value for k'?

(d) We now do code-generation (and making use of the procedure *getreg*) to produce code of the same kind as in the *notat* (from (Aho et al., 1986, Chapter 9)). The intermediate code, for which *machine code* is to be generated, is a basic block containing the following 3 TA-instructions:

```
e = a - b
f = a - c
d = f - e
```

All variables here are ordinary program variables and we assume all of them are live at the end of the block. Different from the situation in the notat, we assume there is *only 1 register* R_0. You may assume that the analysis which gives the *next-use* information, has been done before the code generation starts.

What is the generated sequence of machine instructions? Which machine instruction originates from which TA-instruction. You are not required to give formally the *descriptors*, but write in the comments to the right of the code what the corresponding content of the descriptors are.

□

Solution:

(a) (i) Take as TA-instruction in a block B an assignment to a variable x. This instruction can be removed if the following condition *both* hold

i. the variable is not *used* later in the block.

ii. x is not contained in $outLive(B)$.

(ii) If there is a variable in $inLive B_0$ where B_0 is the *initial block*, then that variable is potentially used before it obtains a value, in one or another execution of the program.

Remark: the answers here are the "expected ones" given the pensum and the formulation of this problem which states that the control-flow graph plus liveness-information for variables is available. Generally speaking, there are other situations, where instructions can safely be removed from a program (it's only that the course did not cover it). "Dead-code" would be an example (i.e., instructions where the control-flow is garanteed never to execute). Note that this is slightly different from the answer given above: there it's about assignment which *are* (possibly) executed, but have no effect whether they are executed or not. Dead code is about *statements* guaranteed not to be executed, dead variables (i.e., non-live variables) is about *variables* which are not used.

For the second question ("initialized variables"): intuitively, one could think of situations where a variable is "declared" but not given a value. That might happen in a high-level language which allows to do that *and* does not specify that in such a situation ("declare-without-define") the variable should obtain a well-defined default value.

However, the problem here does *not* speak about a high-level programming language, but about 3AIC. In this course (and elsewhere), the 3AIC, while not yet being outright machine code (working on registers etc), is rather restricted already and does *not feature* variable declarations! Variable declarations may well be part of the (perhaps high-level) source language, and the 3AIC may well have access to the symbol-table which reflects the scoping rules of the source language. But on the level of 3AIC, there are no variable declarations or lexical scopes *in the program texts*. So answers using those concepts don't capture what is asked here.

(b) Astrid is right. According to the definition of loops from the lecture, neither $\{B_1, B_2, B_4, B_5\}$ or $\{B_1, B_3, B_4, B_5\}$ are loops. For example, the first set of nodes has *two* entry points: B_1 can entered via B_0 (which is not in the "loop"-set), and B_4, which has B_3 as predecessor outside the given set.

Analogously for the second set $\{B_1, B_3, B_4, B_5\}$.

The third given set *is* a loop. And: there is another one, namely the singleton set $\{B_3\}$.

(c) Trivial things first: to move it out of the loop means to move it *before* the loop (not afterwards), obviously. The canonical place thus is *immediately* before the loop we are moving out of. As we are dealing with *loops* in the specific sense discussed (as opposed to general cycles in a graph), there is exactly one well-defined entry point to the loop, and that is exactly where the code needs to be moved to. More precisely, it need to be moved immediately *before* that node. In our example, the entry node of the "big" loop is B_1 and the predecessor outside of the loop is B_0. To place the code, one simply introduces a new block, say B_6, placed *between* B_0 and the loop's entry node B_1. In particular, the code cannot be placed inside B_1 (at the beginning, say)[2] and the arc back from B_5 still has B_1 as successor, and not the new node.

(d) With one register, there's a lot of register-memory traffic

[2]One reason is: in that case it's still part of the loop, which is something we wanted to "optimize". There is a different way of seeing it. If we think that we are not moving code around in a control-flow graph, but actually moving lines in a sequence of TA-instructions (and the control-flow graph is *implict* in the code). In that view, placing the lines directly before the beginning of block B_1 simply does not put them inside B_1, simply by the way the control-flow graph blocks are defined. That placement may well, however, "glue" the new code directly at the end of B_0 without "creating" a new node. Those are rather fine points, introducing a new node in the way described right in front of B_1 is acceptable.

```
 1  //———————————————————— e = a − b
 2  MOV a R0
 3  SUB b R0   // e ∈ r0 , ''all'' reg's full
 4  // ——————————————————— f = a − c
 5  MOV R0 e   // f has a next−use, so, clear
 6             // the only register r0
 7  MOV a R0   //
 8  SUB c R0   // f ∈ r0
 9  // ——————————————————— d = f − e
10  MOV R0 f
11  SUB e  R0   // f is live after the block
12              // and must therefore be saved
13              // f before the SUB step is already
14              // in the right place (in r0)
15              // afterwards, d is in r0
16  //———————————————————— end−of−basic block
17  MOV R0 d    // save value for d back to main memory
18              // all other variables are already up−to
19              // data in their resp. ''home positions''
```

Exercise 7 (Code generation (%))

In this problem we look at *code generation* as discussed in the lecture, i.e., as covered by the "notat" which had been made available and which covers parts of Chapter 9 of the old "dragon book" (*Compilers: Principles, Techniques, and Tools, A. V. Aho, R. Sethi, and J. D. Ullman, 1986*).

(i) Register descriptors indicate, for each register, which variables have their value in this register.

  (i) A single register can contain the values of more than one variable. Give a short explanation/example of how a situation like that can occur. You can keep it really short.

To get more efficient (i.e., faster) executable code, we want to consider transformations of three-address intermediate code, but we restrict ourselves to transformations *local* to basic blocks. We again assume the code generation as done in the "notat"

So assume a basic block consisting of three-address instructions. Those look typically as follows `x` := `y` **op** `z`, where `x`, `y`, and `z` are ordinary variables or *temporaries*. But constants are allowed as well (for instance, as in `x` := `6`), to allow examples with not to many variables.

We consider as the only allowed optmization *to interchange lines* of three-address instructions.

(ii) Describe a *concrete* situation where such an interchange makes the generated code *faster* without of course changing the semantics.

Concrete means, lines of three-address code. Use *one* register only (called `R`). Make all assumptions explicit ("at the beginning of my example, `R` is empty/`R` contains ..."). Explain why the interchange leads to a speed-up, referring to the *cost-model* of the notat/lecture.
□

Solution:

(a) Register descriptors:

  (i) The answer should simply be `x:=y` where `x` and `y` are different variables (resp. have different home positions), or an explanation to that effect. It's not required to give the machine code, an argument suffices. If one does not mention that `x` and `y` are different, it's accepted as ok as well.

We have not looked at the *concrete* code generation *procedure* for the `x := y`. But, it was discussed in the lecture, it's fairly obvious, and it is explicitly mentioned in the notat. It should be immediate.

(b) *Local optimization:* It should be fairly easy to figure out one example covering at least the *spirit*. To get a speed-up, we need to avoid *register-memory traffic*. One can different points of the code generator to illustrate the speed-up.

For a correct answer, one should give

- original 3AC program plus clear indication of what is swapped
- the generated machine codes resp. the generated machine code from the original and explain what changes and why
- mention how that affects the costs in the cost model. Exact calculation of the given "program" is not needed, but reference to the cost model is.

The code generation has some fine points (like liveness etc). For a full answer, let's not insist on that.

0.0.0.1 One example: "purging" a/the register

In the cost model (and in general) register-memory traffic costs. Especially it costs *more* than operations on registers. The idea of an example is therefore: before the swap, the only register is being used for one step of the code, after the swap, it cannot be used for that step, as it's being used for something else. That requires that the value has to be stored back to the home position and reloaded later. That makes the program "more costly".The example from Listing 9 and 10 makes use of that.

Listing 9: Reuse of a register for y

```
 1    // initially , R empty
 2  ------------------------------------------------------------------
 3    y := x + 1    // use R for the result:
 4                  // Load x                                        1
 5                  // R -> y (not up-to date)
 6    z := y + 1    // re-use R (containing y): 0 Reg-Mem move       0
 7                  // for loading it. So, (2) of code-gen omits
 8                  // the MOV
 9                  // however: y needs to be saved (which
10                  // is required by get-reg, case (3)
11                  // Store y (because it's assumed to be live )   1
12                  // R -> z  (not up-to date)
13    a := t1 + t2  // Store R z (save z)                            1
14                  // load t1                                       1
15                  // load t2                                       1
16                  // R -> A  (not up-to date)
17  ------------------------------------------------------------------
18                  // end of block: save a                          1
```

Listing 10: Reuse of register no longer possible

```
 1    // initially , R empty
 2  ------------------------------------------------------------------
 3    y := x + 1    // use R for the result:
 4                  // Load x:                                       1
 5                  // R |-> y (not up-to date)
 6    a := t1 + t2  // Store R -> y    (get-reg -(3)                 1
 7                  // Load t1                                       1
 8                  // Load t2                                       1
 9                  // R |-> a  (not up-to date)
10    z := y + 1    // Store a (no reuse)                            1
11                  // Load y                                        1
12                  // result: R <- z (not up-to date)
```

```
13
14                    // end of block: store z                    1
15
```

□

References

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