Universitetet i Oslo
Institutt for Informatikk

Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2022 **Series 8** 19. 4. 2022

Topic: Code generation (Exercises with hints for solution)

Issued: 19. 4. 2022

The exercises here are mostly taken from older exams. More exam questions, including more questions involving code generation are uploaded in a document `examcollection.pdf` (in the directory `handouts`).

Table 1 shows from which exams the exercises are taken.

Exercise 2	exam 2007, problem 4(a)
Exercise 3	exam 2009, problem 4
Exercise 4	exam 2010, problem 4(d)
Exercise 5 (3) + (4)	exam 2011, problem 4(c)+4(d)
Exercise 6	exam 2016, 4
Exercise 7	exam 2017, 6

Table 1: Exam questions

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...])

1. This is meant 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.

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

3. 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:

1. 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).

2. 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) and 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.)

3. 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 (-%))

1. 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), i.e. to simply list the variables for which the info is needed (a, b, c, d) and state their liveness status + some words of explanation (in an exam, one can keep the explanations shorter than the ones here).

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: The 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
```

1. 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.

2. 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.

3. 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.

4. 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.

5. 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:

1. 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.

[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.

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
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 ------------------------------
```

2. For the CFG. see below in e)

3. A possible rule could be

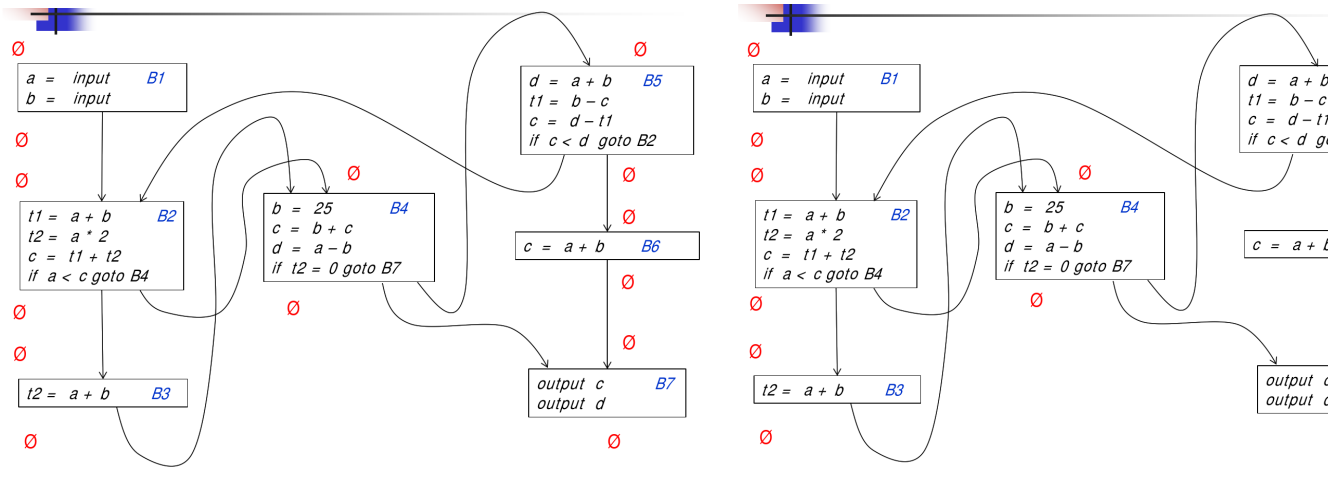
   > All temporaries which are *used* in a given basic block must be assigned to ("defined") in the same before the (first) use.

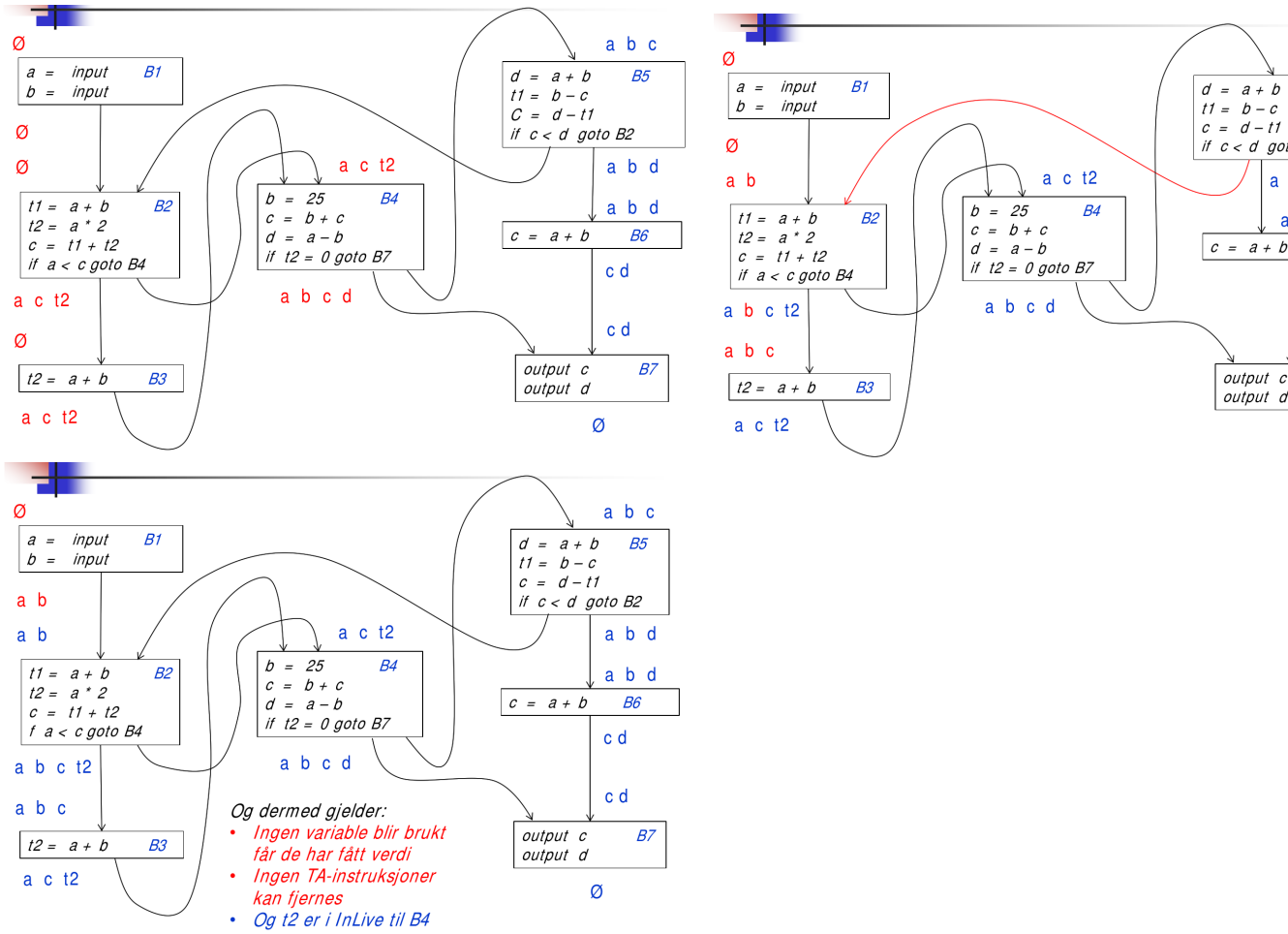
   Another way of saying it is:

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

4. sanitary check: In block B_4, the temporary t_2 violates the formulated rule.

5. Liveness:





Exercise 4 (Code generation (–%))

1. 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: check whether 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 the 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 2: P-code instructions

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

(a) 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.

(b) Sketch which *data structures*

2.

3. 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 whether the user-level source language assures an *order* of evaluation of compound expressions. Other factors you think relevant can be discussed as well.

4. 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:

1. (a)

   (b)

   (c)

2.

3. The following is from the given solution at that time.

   (a) 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.

(b) The situation leaves room for many optimizations. One situation discussed 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.

4. 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 (%))

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*).

1. Register descriptors indicate, for each register, which variables have their value in this register.

   (a) 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.

2. 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:

   (a) 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.

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 ------------------------------------------------------------------
```

□

Exercise 7 (Code generation (%))

(a) Consider the following transformation on three-address code, illustrated on the following example.

Listing 11: Before

```
1 if t == 0
2 then
3    x = y + z;
4    <rest of then-branch>
5 else
6    x = y + z;
7    <rest of else-branch>
8 endif
```

Listing 12: After

```
1 x = y + z;
2 if t == 0
3 then
4    <rest of then-branch>
5 else
6    <rest of else-branch>
7 endif
```

The idea is to move "common instructions" (like the assignment `x = y + z` in the example) *before* the conditional, so long it does not change the semantics of the code. The three address code in this sub-problem supports two-armed conditionals (if-then-else), not the if-goto constract as in the lecture and in sub-task (b).

Assume code generation as covered in the "notat" 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*). Assume further that the code generator has access to *local* liveness information, i.e., liveness information per basic block, but *no global liveness* information is available.

Under these assumptions, what are potential effects of the code transformation on the qualitity of the generated code? Discuss this question referring to the cost model of the notat/lecture.

Note: neither exact sequences of possibly generated two-address code nor detailed calculations of costs are expected/needed as answer, just a short discussion of the influence of the transformation on factors of the cost model.

(b) Consider the program from Listing 13 in three address code. We do not distinguish here between temporaries and standard variables.

(a) Indicate the *basic blocks* in giving start and end line for each block (numbering the blocks like B_0, B_1, etc.) You can also use the code repeated in the appendix, drawing clearly visible horizontal lines indicating the boundaries of the blocks and give the B_i-numbers of the blocks.

(b) Draw the control flow graph of the program using B_0, B_1 from the previous question to identify the nodes of the graph.

(c) Does the control-flow graph contain a *loop?* Use the notion of loops for control-flow graphs from the lecture.

(d) Give the *inLive* and *outLive* information for each block (best in the form of a table).

Listing 13: Three-address code

```
1  x = input
2  y = input
3  label L1
4  b = x + y
5  z = b*z
6  label L2
7  x = a + 1
8  if_false x goto L3
9  x = y + x
10 if_true z goto L5
```

```
11 goto L1
12 label L3
13 z = b * 2
14 goto L2
15 label L5
16 output x
```

Solution:

(a) The task gives only 8 points and no huge calculations or deep or long essays is expected. There is also no unique best answer, the stituation is at least *ambiguous*. Factors that influence the effect of that transformation are the following:

- memory-register traffic costs.
- since we have block-local information only, a *variable* (not a temporary) at the end of a block is considered *live*.
- temporaries are stored back at the end of a block (they are treated as if they were dead). The latter is the way temporaries are supposed to be treated by the code generation.

All these are factors for an answer. One good answer could mention: if one moves an assignment in front of such a two-armed conditional (in the way sketched by the illustrative example): variables that occur in there (like y) are then definitely considered live (as they occur *at the end* of the block before the branching. Therefore, if they happen to be in a register, the register cannot be freed [check that, maybe they have to moved back to the home position]. By "freeing" I mean the last step in the code generation, see the slides around slide 55 about "recycling registers" using liveness information.[2] If in term instead is in the branches (before the optimization), it can be that in the *rest* of that basic block, y (for instance) is not used any more, i.e., it's clear that it's not live. Therefore, the code generator knows now *locally* that y is dead and is able to register. With an additinal register free, that *can* improve the performance (avoiding memory-register traffic for other variables etc).

The above explanation used the transformation putting $x = y + x$ right in front of the block. It's only an example, if one uses differnent such transformations (pulling a line or even more) in front, that's ok too.

Instead of concentrating on y (or z), one can also take x for an argument. See the *getreg*-algo, which determines the location where x is supposed to stay. For instance see slide 59 and around there. x will be put into a register, if there is one free. If not, *liveness* information for x will be taken into account, resp. next use in the block (in case 3).[3] In case there is a next use, x can end up in a register (and if one is lucky, the content of that register does not need to be written back!). If one moves the assignment in front, then the condition does no longer apply, and x needs to be written back to main memory.

One particularly neat (and detailed/insightful) argument could take into account x and y, where y is loaded to a register and y is dead (see point 1 of *getreg*). There, one can simply reuse the register for y. If one becomes "considered life", point 1 does no longer apply, so in the worst case 4 applies, so then x has to be stored back immediately. Note that in case 4, the storing back is done unconditionally. Keeping it in a register may save memory-traffic: if a register value and a home location diagree, "reconciling" them is avoided by the code generation for dead variables, when dead means: the next thing that happens is overwriting them.

A *non-argument* is: the code gets *shorter*, therefore the cost model says better. It's true that the code gets shorter. But the question is not the space is main-memory, but how

[2] That the register for, say y, can be freed would assume additionally, that the register contains an up-to date value of y (and would contain not the value for other non-dead variables on top). If the value in y's "home location" is stale and *if* the register is freed, then of course the register would have to be written back, before freeing it. We don't expect an argument on this level of detail, especially since the details of which register to take when purging a register were not fully given in the book.

[3] If x is dead, then there's no need to actually do the assignment. However, the code generation does not take that into account. It will not omit the assignment to a dead x. A more clever algo might do that though.

many "lines of code" will have to be loaded to the processor at run-time. Therefore the code for $z = y + z$ when given in *both branches* is nonetheless only executed one depending on which branch is taken.[4]

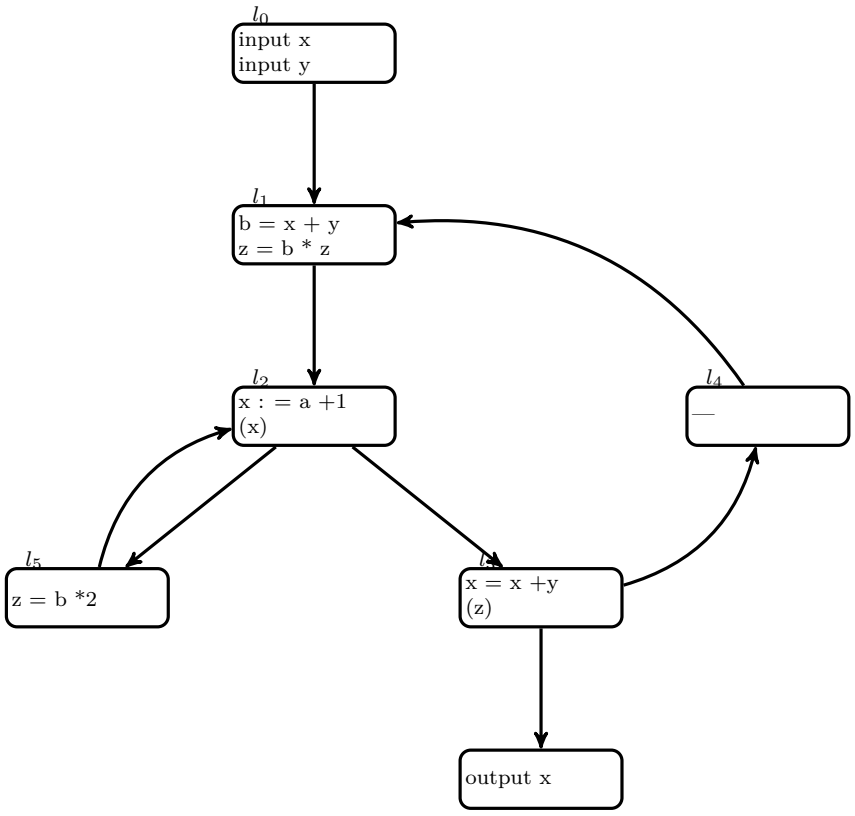
(b) This one should be rather standard. Finding the basic blocks should be faster than the other, the second one is more cumbersome (but also standard). I weight 6 (blocks, CFG, loop) vs. 8 (inlive and outlive).

In the suggested solution from Listing 14, I added a block (without name) which contains only one goto-statement. One can make the argument that this is a block. One may also see it as node that is empty (it contains no real code, just a jump which should be represented ultimately as *edge* in the cfg.) Wheter or not a node is drawn in the CFG representing that block does not matter for the correctness of a solution. In my solution, I did not bother to draw it. Being an empty node, its inlive and outlive would coincide.

[4]Side remark: the code generator might of course generate *different* 2AC from $x = y + z$ in each of the two branches.

Listing 14: Three-address code, BBs indicated

```
1  ---------- B0 ----------
2  x = input
3  y = input
4  ---------- B1 ----------
5  label L1
6  b = x + y
7  z = b*z
8  ---------- B2 ----------
9  label L2
10 x = a + 1
11 if_false x goto L3
12 ---------- B3 ----------
13 x = y + x
14 if_true z goto L5
15 ----------B4 ----------
16 goto L1
17 ---------- B5----------
18 label L3
19 z = b * 2
20 goto L2
21 ---------- B6----------
22 label L5
23 output x
24 ------------------------
```



(c) A good answer should take into account the fact that there is only *local*

	L_{in}	L_{out}
B0	$\{a, z\}$	$\{a, x, y, z\}$
B1	$\{a, x, y, z\}$	$\{a, b, y, z\}$
B2	$\{a, b, y, z\}$	$\{a, b, x, y, z\}$
B3	$\{a, x, y, z\}$	$\{a, x, y, z\}$
B4	$\{a, x, y, z\}$	$\{a, x, y, z\}$
B5	$\{a, b, y\}$	$\{a, b, y, z\}$
B6	$\{x\}$	$\{\}$

□

Korrektion description:

- An answer that shows one has understood the code generation and the cost-model and gives a reasonable explanation (perhaps an example) would get full points. I expect that many would take the example with $x = y + z$ as basis for an argument (which is fine, but not required).

  Often, the answer was skipped. In general, it was not answered very well.

- For the global analysis, the blocks, the graph, and the loop question were answered ok. Each gave 2 points. The liveness information: the answers were mixed, it was below average.

□

References

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