UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in	INF5110 — Kompilatortteknikk
Day of examination:	7. June 2017
Examination hours:	09.00 – 13:00
This problem set consists of 22 pages.	
Appendices:	3 pages
Permitted aids:	All written and printed

Please make sure that your copy of the problem set is complete before you attempt to answer anything.

- You should read the whole problem set before you start, getting an overview can help to make wise use of the time.
- Besides writing in a readable manner, draw requested figures in a clear way.
- Give concise and clear explanations!
- You may answer Problem 4, 5, and part of Problem 6b by filling in the pages in the appendix and hand them in together with the rest of the answers (in the "white version").

Good luck!

Exam questions + hints for a solution

(Continued on page 2.)

Problem 1 Compiler front-end phases (weight 8%)

The front-end of a compiler contains typically the following phases: the lexical phase (= scanner), the syntactic phase (= parser), and the semantic analysis phase. The phases check and process elements of the language being compiled in particular ways. For each of the following "language rules", specify which of the three mentioned phases is best suited to check compliance. If a check can (reasonably) be done in more than one phase, shortly give arguments for trade-offs involved.

(i) A function has to be called with the *correct number of arguments.*

(ii) Underscore characters "`_`" are allowed in the middle of identifiers, but not at the beginning or the end (i.e. "`my_id`" is legal but "`_id`" is not).

(iii) Every variable must be declared *before* it is used.

(iv) Assignment statements must end with a semicolon "`;`".

Solution:

(i) **Semantics.** One has to compare the declaration and the call of the function. That cannot be done by the parser.

(ii) **Scanner.** There is no reason to postpone that. It is a very "standard" thing for a scanner to do (as one can see by the fact that it's possible to write regular expressions capturing this condition). Identifiers (with restrictions such as the one mentioned) are typical tokens (which are the output of a scanner and the input of the parser).

(iii) **Semantics**, probably. At least that's nowadays a probable arrangement: the parser gives back the AST, and the semantic phase can check it traversing the AST one time (maybe making use of a symbol table, where a declaration adds info about a variable into the table, and use of the variable tries to look it up, raising an appropriate error if the look-up fails. In terms of AGs, being defined most probably would be an inherited attribute (not part of the question, though).

It is possible do make the check by the parser as the parser will parse the code from "beginning-to-end". So the parser can maintain information (like said symbol table) and do the "declare-before-use" check.

There's no particular advantage doing that though (perhaps one may save one traversal, in that it's done while parsing, perhaps in a very *simple* language, that's all what is needed).

(iv) **Parser.** It's *not* possible by the scanner, being a statement —variable on the left, some compound expression or some other compound

(Continued on page 3.)

syntactic element on the right— is a notion covered and defined by the parser. The scanner does not know if a semicolon is used in the context of a assignment statement.[1]

□

Remarks how to correct/how correction was done: 4 subitems for the 4 questions, 2 points each. Without explanation, but correct, full points. So it's mostly hit-or-miss for each of the 4 questions. In case there is 2 possible phases (mostly for the variable declaration question), if one phase ok, the other not (or justification wrong), only 1 point for that one. Note: if 2 phases were given, the question required an explanation.

It was intended as warm-up, and overall, the answers were given correctly, or almost fully correct in a high degree. □

[1] It may be theoretically conceivable to have simplistic language (like straight-line code consisting *only* of a list of assignments), where the scanner can do that. That's an unlikely answer.

(Continued on page 4.)

Problem 2 Regular expressions (weight 12%)

Remote file identifers look, in the most general, as follows:

```
user@hostname:filename
```

More precisely: The parts of the identifiers are made up of *words*, which are sequences of one or more *letters* and *digits*. The *user* part contains a *single* word. A *hostname* consists of one or more words separated by *periods* (as in `www.uio.no`). A *filename* consists of one or more words separated by slash characters ("/") with an optional leading and/or trailing slash. The "`user@`" part is optional and may be omitted. The entire "`user@hostname:`" may also be omitted, including the trailing colon "`:`". The "`user@`" part may not appear unless the "`hostname:`" part is also present.

Now: specify the form of remote file identifiers using *regular expressions.* Feel free to make use of the more "user-friendly" versions of the regular expressions.

Solution:

$$word = [a-zA-Z0-9]^+$$
$$((word\,\texttt{@})?\,word(\texttt{.}\,word)^*\,\texttt{:})?(\texttt{/})?\,word(\texttt{/}\,word)^*(\texttt{/})?$$

□

Remarks how to correct/how correction was done: One can split it into user, host, and filename and the global optionalities, 3 point max each. Possible expected errors could be that the word is taken to start with a letter (but the text of the exam question does not require that).

We expect that things like *word* or *letter* are spelled out (not just appealing to general understanding of what letters are or def's from the lecture).

(Continued on page 5.)

Problem 3 Top-down parsing (weight 24%)

Consider the following context-free grammar:

$$\begin{array}{lcl} S & \rightarrow & ABA \\ A & \rightarrow & B\mathbf{c} \mid \mathbf{d}A \mid \boldsymbol{\epsilon} \\ B & \rightarrow & \mathbf{e}A \end{array} \tag{1}$$

3a First- and follow set (weight 10%)

Give the first- and follow-set for the non-terminals of the grammar.

3b LL(1)-parsing (weight 14%)

Give the LL(1) parsing table for the grammar. Is the grammar LL(1)?

Solution:

(i) The solution is given in Table 1.

	first	*follow*
S	$\mathbf{d}, \mathbf{e}$	$\mathbf{\$}$
A	$\mathbf{d}, \mathbf{e}, \boldsymbol{\epsilon}$	$\mathbf{\$}, \mathbf{c}, \mathbf{d}, \mathbf{e}$
B	$\mathbf{e}$	$\mathbf{\$}, \mathbf{c}, \mathbf{d}, \mathbf{e}$

Table 1: First and follow sets

Computing the first- and follow-sets is a completely standard task. Nullable information ($\boldsymbol{\epsilon}$) for the first-set and the end-of-parse marker ($\mathbf{\$}$) for the follow-set must be present for a complete answer. The only case which is "tricky" (which might be overlooked if doing it carelessly) is perhaps the $\mathbf{c}$ (and perhaps also the $\mathbf{d}$) in the follow-set of A. The follow set of B is easier, but one has to take into account that A is nullable (as seen in A's first-set).

(ii) The solution is given in Table 2. It's obviously *not LL(1)* due to double entries in the slots.

The calculation of the LL(1) table is described in [1], also on the slides (the "table recipe" approx. on slide 189 in the corresponding section). The tables in the book and the slides would copy in the rules literally into the slots of the table. I assume the rules are numbered (from 1 to 5 in the order given), and I use just those numbers (accepable as solution as well if the numbering is clear).

(Continued on page 6.)

	c	**d**	**e**	**$**
S		1	1	
A	4	3, 4	2, 4	4
B			5	

Table 2: LL(1) parsing table

For a better overview, here's a numbering of the productions plus the calculation of relevant information per rule (first-set of the right-hand side, whether the right-hand side is nullable[2] and the follow-set of the non-terminal on the left-hand side).

```
                    first of RHS    RHS nullable?  Follow LHS
-------------------------------------------------------------
  1:   S -> ABA     e,d               -            [no need]
  2:   A -> Bc      e                 -            [no need]
  3:   A -> dA      d                 -            [no need]
  4:   A -> eps     esp               +            $,c,d,e
  5:   B -> eA      e                 -
-------------------------------------------------------------
```

With this information, the required Table 2 is easy to be filled out. The blue entries correspond to "condition one" for the table, adding a rule based on the first-set. The red entries correspond to "condition two", adding entries for the left hand side for a nullable rhs.

□

Remarks how to correct/how correction was done:

(i) (i) For a): The follow set is (as always) harder to calculate than the first set (especially due to ϵ). I weigh 4 vs. 6 points.

(ii) If the previous task resulted in a *wrong* follow set, but filled the table here "relatively correctly" based on that, I will give (almost) all points in this sub-problem. I will not give 100% full points, as the second task does not 100% depend on the first (insofar that first-sets have to be calculated here again, namely for the right-hand sides). Note that a column for **$** is required for a complete solution.

(ii) For the table, also the form of the table should be LL(1) (not SLR(1) or something else).

[2]Technically, this collumn can be seen as implied by the first-set, namely by our conventions, ϵ in the first set represents nullability.

(Continued on page 7.)

The averaged level of answers was average, not too high variants (but the question was pretty standard, especially the first- and follow sets. Mayb drawing the parsing table was less often (mostly we did LR-tables in the exercises/earlier exams).

□

(Continued on page 8.)

Problem 4 Bottom-up parsing (weight 20%)

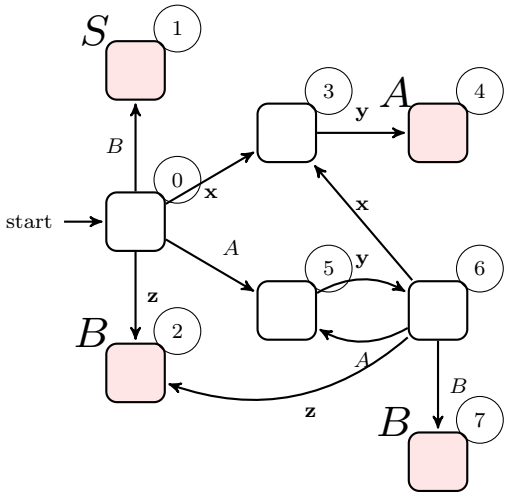


Figure 1: LR(0)-DFA (items missing inside states)

4a LR(0)-DFA (weight 14%)

Assume a context-free grammar with non-terminals S (the start symbol), A, and B, and terminals **x**, **y**, and **z**. Consider the LR(0)-DFA in Figure 1. Shown are all states and transitions, with the start state 0. Furthermore indicated are states, which contain a *complete item,* i.e., an item with the "**.**"-marker at the very end: For states containing such a complete item, the corresponding non-terminal on the *left-hand side* of the complete item is indicated, as well. For instance, the state numbered 1 is assumed to contain a complete item with S as left-hand side of the production, i.e., an item of the form

$$S \rightarrow \text{<right-hand-side>} \ \textbf{.}$$

Correspondingly for states 2,4, and 7.

Now, fill in the missing information, i.e., the items that form the states 0 – 7, so that the filled-out DFA is the result of the standard LR(0)-DFA construction of the grammar (by filling out the items correctly, you implictly reconstruct the grammar, as well).

The automaton is reproduced in the attachment, which you can use for your solution. It's advisable to make a sketch first on a separate sheet, to copy it in (readably) afterwards.

4b Classification (weight 6%)

For the grammar from problem 4a: is it LR(0), LR(1), SLR(1), LALR(1)?

(Continued on page 9.)

Solution:

LR(0)-DFA's as such are standard. In principle the task should actually be easy. The task, however, is formulated with a twist, in that one does not have to construct the DFA from the grammar, but "reconstruct" the grammar from the automaton, basically.

Parts of it should be really straightforward (the simple productions), but one also needs to remember how to do the *closure* for the states of a LR(0)-DFA.

In state 6, one have to have the closure correct. Assuming that one has figured out that an item $B \rightarrow A\mathbf{y}.B$ is "contained" in state 6, then adding the closure is conceptually done in two steps. First the closure for B, and then the one for A. The outgoing edges from 6 also indicate that there are (at least) 4 items to be expected in this state.

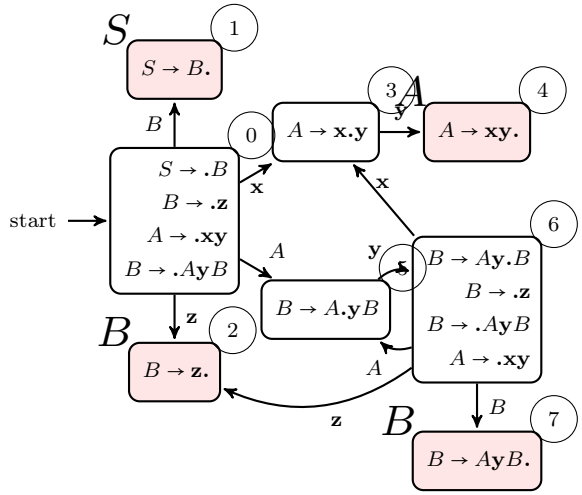


Figure 2: Solution LR(0)-DFA

□

Remarks how to correct/how correction was done: This one was answered correctly to a very high degree. When making the exam I was afraid that it was too "hard" as it perhaps would look unclear what has to be done (due to novelty of posing the problem). That was unfounded. It turned out that the task was easy enough (and if so, got too many points perhaps).

If someone does not forgot the closure right (for state 6), I downgraded already 50% as that is one half of the general setting of how those LR(0)-automata work. Making an error in the closure was graded less harsh.

The task was in general also very easy to correct (with the appendix automaton to fill out). □

(Continued on page 10.)

Problem 5 Attribute grammars (weight 16%)

Consider the following grammar.

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textbf{prog}\ \textit{stmt-seq} \\
\textit{stmt-seq} & \rightarrow & \textit{stmt}\ \textit{stmt-seq} \\
\textit{stmt-seq} & \rightarrow & \textit{stmt} \\
\textit{stmt} & \rightarrow & \textbf{do var = const upto const begin}\ \textit{stmt-seq}\ \textbf{end} \\
\textit{stmt} & \rightarrow & \textbf{assign}
\end{array}
$$

It describes a (very simple) language, which allows to iterate through sequences of statements, ultimately assignments. The terminals are given in **boldface**, the non-terminals in *italics*. A **do**-loop is specified by the range of the loop variable, given by integer constants, representing the loop's lower and upper bound. No production for assignments is given (as irrelevant for the task). Thus, assignments, represented by **assign**, are treated here as terminals.

A sample program is given in Listing 1 (i is the token value for **var**, similarly for 1 and 42).

Listing 1: Sample program

```
1 prog
2   do i = 1 upto 42
3   begin
4      assign
5      assign
6    end
7    assign
```

Write an attribute grammar that determines for each assignment

> how many times the assigmment will be executed

when running the program. Assume that the terminals representing constants have an attribute `val` denoting their constant integer value. The result should be found in an attribute for the **assign**-symbol, say `iterated` or `i` for short. You may make use of that attribute for other symbols of the grammar, as well, as needed for your solution.

Assume that for each loop the lower bound $\textbf{const}_0$ is smaller or equal the upper bound $\textbf{const}_1$ (no need to check that in your solution). Make sure you calculate the required attribute value *exactly* (not more or less correct, plus/minus one).

(Continued on page 11.)

	productions/grammar rules			semantic rules
0	*program*	→	**prog** *stmt-seq*	
1	*stmt-seq*$_0$	→	*stmt stmt-seq*$_1$	
2	*stmt-seq*	→	*stmt*	
3	*stmt*	→	**do var** = **const**$_0$ **upto const**$_1$ **begin** *stmt-seq* **end**	
4	*stmt*	→	**assign**	

Give your answer by filling out the given table. You may use the corresponding form in the appendix (by tearing it out and deliver it with the "white sheets").

Solution: Here's a possible solution. Crucial parts are the treatment of the loop, obviously. Also that one starts at the top-level (i.e., for **prog**) with 1. In general, the general works with *inherited* attributes. It's not really necessary that both **prog** and the top-level statement sequence carry `i` = 1, there is some variations possible. Important is, that one stops at the root of the tree "on top" and pushes the information down to the leaves, i.e., to the attribute `i` for the **assign**. Similarly the "..." in the treatment of the production for the loop. Whether or not one adds the attribute `i` to, for instance **var**, is irrelevant for a correct solution. Note that the tasks requires that the **assign** carries the attribute `i`, for the other elements, it's not required, that all of them carry it, so **prog** may or may not have it.

(Continued on page 12.)

	productions/grammar rules	semantic rules
0	*program* → **prog** *stmt-seq*	*program*.i = 1
		prog.i = 1
		stmt-seq.i = 1
1	$stmt\text{-}seq_0$ → *stmt* $stmt\text{-}seq_1$	*stmt*.i = $stmt\text{-}seq_0$.i
		$stmt\text{-}seq_1$.i = $stmt\text{-}seq_0$.i
2	*stmt-seq* → *stmt*	*stmt*.i = *stmt-seq*.i
3	*stmt* → **do var** =	*stmt-seq*.i = *stmt*.i × (**const**$_1$.val − **const**$_0$.va
	const$_0$ **upto const**$_1$	...
	begin *stmt-seq* **end**	
4	*stmt* → **assign**	**assign**.i → *stmt*.i

□

Remarks how to correct/how correction was done: This question had the biggest variance in the answers so far (quite a few with full points, quite some 0 points, either by not filling it out, or else having a more or less meaningless solution (not just wrong)). Few in between. For that reason the task was easy to correct (with exception of 1 "strange but not 100% wrong" answer). If the boundaries were wrong (something which was explicitly required): 3 points off.

The question was not answered particularly well (seen the average). □

(Continued on page 13.)

Problem 6 Code generation (weight 20%)

6a Code generation and optmimization (weight 8%)

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

Listing 2: 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 3: 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 6b.

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.

6b Global analysis (weight 12%)

Consider the program from Listing 4 in three address code. We do not distinguish here between temporaries and standard variables.

(i) 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.

(ii) Draw the control flow graph of the program using B_0, B_1 from the

(Continued on page 14.)

previous question to identify the nodes of the graph.

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

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

Listing 4: 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
```

(Continued on page 15.)

Solution:

(i) 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 dead (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. By "freeing" I mean the last step in the code generation, see the slides around slide 55 about "recycling registers" using liveness information.[3] 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).[4] 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!).

[3] 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.

[4] 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.

(Continued on page 16.)

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 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.[5]

(ii) 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).

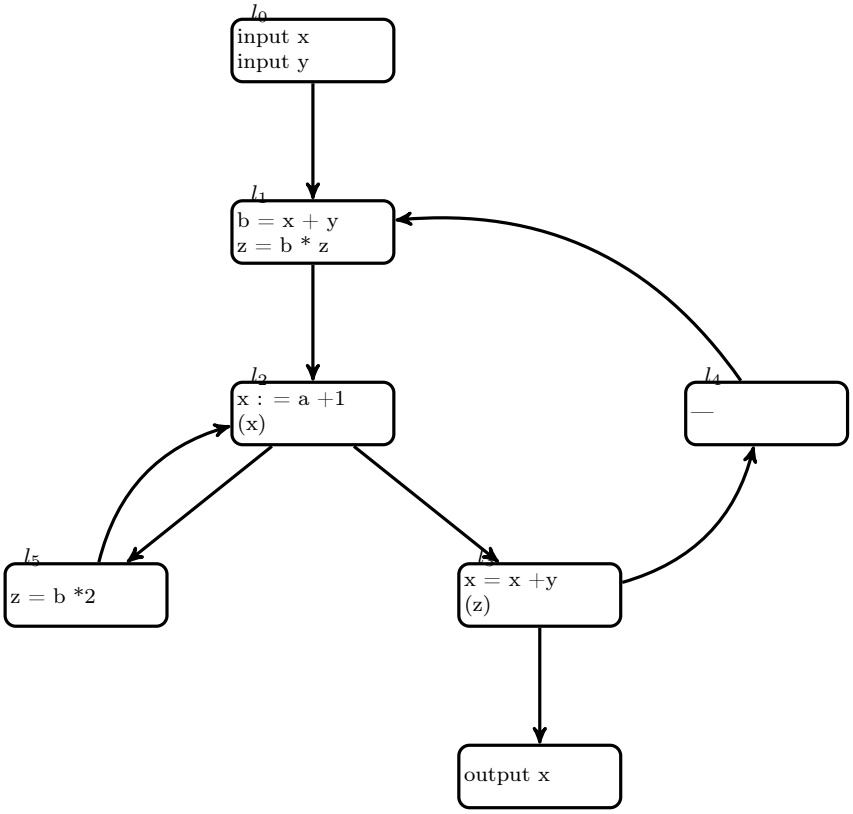
In the suggested solution from Listing 5, I added a block (without name) which contains only one goto-statement. One can make the argument that this is a block. One may also see it as node that is empty (it contains no real code, just a jump which should be represented ultimately as edge in the cfg.) 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.

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

(Continued on page 17.)

Listing 5: Three-address code, BBs indicated

```
--------- B0 ----------
x = input
y = input
--------- B1 -----------
label L1
b = x + y
z = b*z
--------- B2 -----------
label L2
x = a + 1
if_false x goto L3
------- B3 ---------
x = y + x
if_true z goto L5
--------B4 -----------
goto L1
--------- B5---------
label L3
z = b * 2
goto L2
--------- B6-------------
label L5
output x
-------------------------
```



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

(Continued on page 18.)

	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	$\{x\}$	$\{\}$

□

Remarks how to correct/how correction was done:

- 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 answered.

- 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.

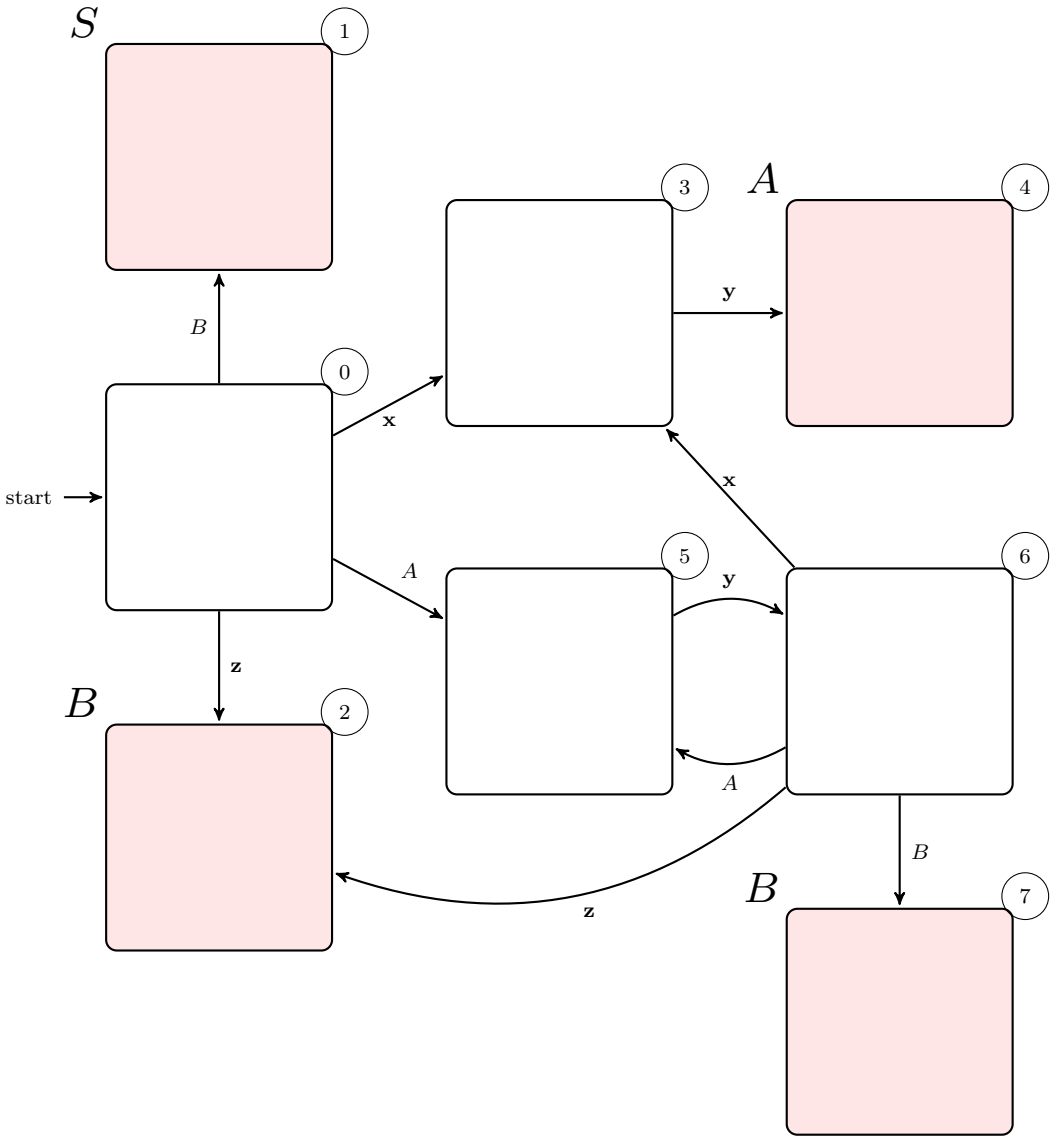
□

(Continued on page 19.)

Appendix: DFA for Problem 4

Candidate nr.: .....

Date: .....



(Continued on page 20.)

Appendix: Form for Problem 5

Candidate nr.: .....

Date: .....

	productions/grammar rules	semantic rules
0	*program* → **prog** *stmt-seq*	
1	$stmt\text{-}seq_0$ → *stmt* $stmt\text{-}seq_1$	
2	*stmt-seq* → *stmt*	
3	*stmt* → **do var =** $\mathbf{const}_0$ **upto** $\mathbf{const}_1$ **begin** *stmt-seq* **end**	
4	*stmt* → **assign**	

(Continued on page 21.)

Appendix: The code for Problem 6b

Candidate nr.: .....

Date: .....

```
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
```

(Continued on page 22.)

References

[1] K. Louden. Compiler Construction, Principles and Practice. PWS Publishing, 1997.