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 12 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!

(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 "`;`".

(Continued on page 3.)

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.

(Continued on page 4.)

Problem 3 Top-down parsing (weight 24%)

Consider the following context-free grammar:

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

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)?

(Continued on page 5.)

Problem 4 Bottom-up parsing (weight 20%)

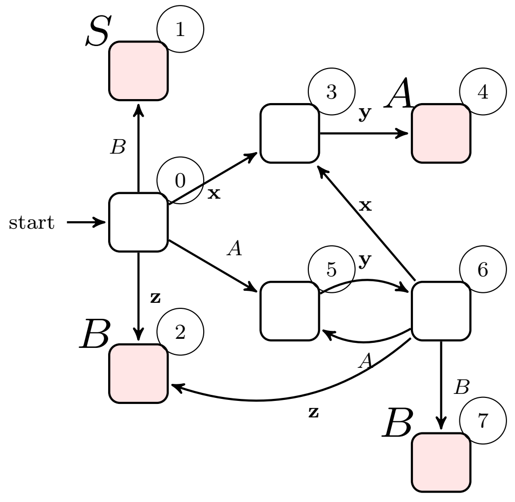


Figure 1: LR(0)-DFA (items missing inside states)

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

$$S \to \text{<right-hand-side>} \;.$$

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 6.)

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

```
prog
  do i = 1 upto 42
  begin
     assign
     assign
   end
   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 7.)

	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**	

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").

(Continued on page 8.)

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

```
if t == 0
then
   x =  y + z;
   <rest of then-branch>
else
   x = y + z;
   <rest of else-branch>
endif
```

Listing 3: After

```
x =  y + z;
if t == 0
then
   <rest of then-branch>
else
   <rest of else-branch>
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 9.)

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

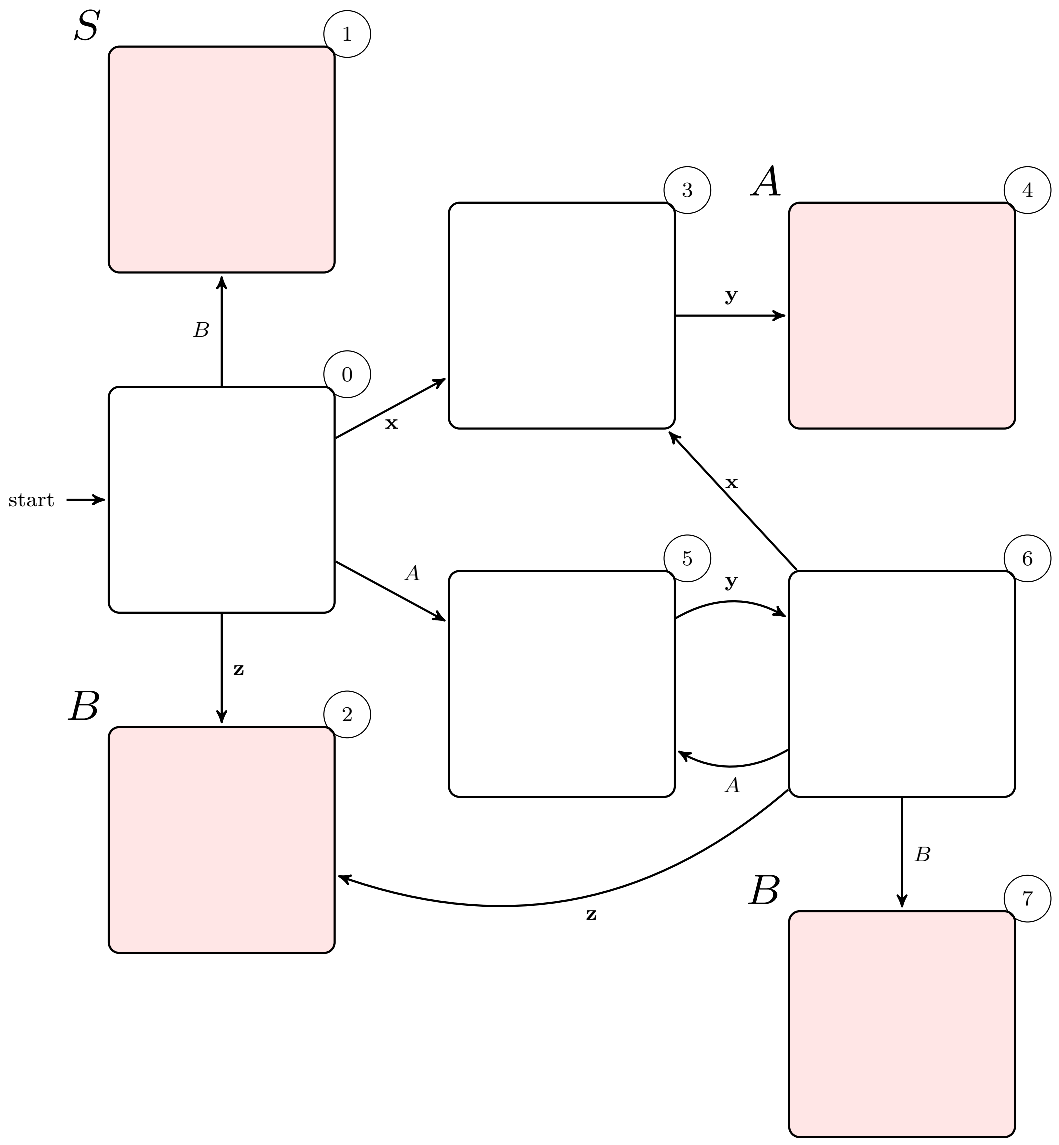
```
x = input
y = input
label L1
b = x + y
z = b*z
label L2
x = a + 1
if_false x goto L3
x = y + x
if_true z goto L5
goto L1
label L3
z = b * 2
goto L2
label L5
output x
```

(Continued on page 10.)

Appendix: DFA for Problem 4

Candidate nr.: ....

Date: ....



(Continued on page 11.)

Appendix: Form for Problem 5

Candidate nr.: ....

Date: ....

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

(Continued on page 12.)

Appendix: The code for Problem 6b

Candidate nr.: ....

Date: ....

```
x = input
y = input
label L1
b = x + y
z = b*z
label L2
x = a + 1
if_false x goto L3
x = y + x
if_true z goto L5
goto L1
label L3
z = b * 2
goto L2
label L5
output x
```