UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in	INF5110 — Kompilatortteknikk
Day of examination:	8. Juni 2016
Examination hours:	14.30 – 18.30

This problem set consists of 17 pages.

Appendices:	None
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.
- Problems 1 – 4 are basically independent from each other. For the *subproblems* of one problem, it is often advisable to tackle them in the order stated.
- Besides writing in a readable manner, draw requested figures in a clear manner.
- Give concise and clear explanations!

Good luck!

(Continued on page 2.)

Problem 1 Regular expressions and scanning (weight 24%)

1a Regular languages (weight 7%)

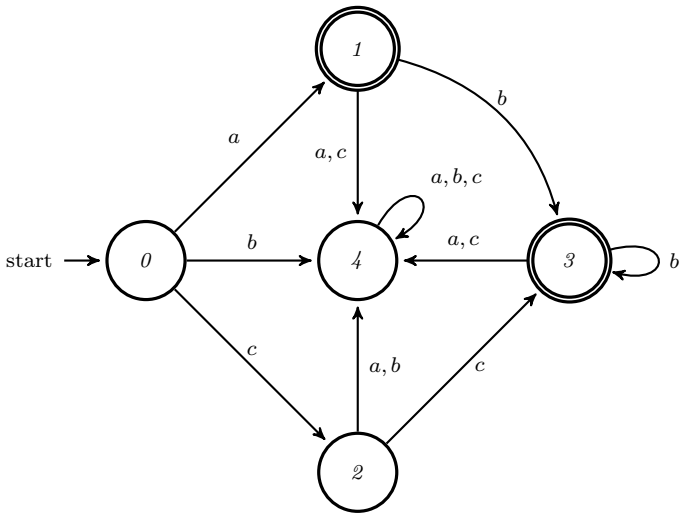
Let Σ be a non-empty finite alphabet, otherwise left *unspecified.* Consider the following language:

$$\mathcal{L} = \{ww \mid w \in \Sigma^*\},$$

in other words: all strings repeating a word over Σ *two times* in a row. Is the language *regular* or *not?* If the language is regular, give a regular expression capturing the language. If not, give a short argument, explaining why not. Are there special cases where the answer would be different from the general case?

1b Minimal automaton (weight 7%)

Is the following automaton *minimal*? Give a short explanation. You may make use of the *minimization algorithm* or, alternatively, give a short explanation clarifying the situation.



1c "C-style" comments (weight 10%)

The task here is to specify a regular expression for "C-style" comments. To notationally (but not conceptually) ease the task, we make the following simplifications compared to the normal situation for C comments:

(Continued on page 3.)

The *alphabet* for our special version of the "C-language" consists of the following 3 symbols

$$\Sigma = \{z,\ o,\ /\}$$

- Arbitrary alphanumerical symbols are represented by z, o, and `/` ("slash").
- Comments here are not delimited by `/* .... */` as in C, but by `/o ... o/`. This is simply done to avoid confusion with the regular-expression star-operator when doing a handwritten solution.

So, comments are *delimited* by "`/o`" and "`o/`".

> More precisely: a comment *starts* with the two symbols "`/o`" and ends with the *first subsequent* "`o/`". For the task, the delimitors *slash-o* and *o-slash* are part of the comment. Comments *cannot* be nested.

Note:

- It is allowed that "`o`" and "`/`", and also "`/o`" occur inside a comment.
- "`/o/`" is not a comment, but "`/oo/`" and "`/o/o/`" are.

Give a single regular expression that matches the comments specified as above.

Solution:

(i) The special cases would be whether Σ has 2 or more symbols, or less. Left out is the case of $\Sigma = \varnothing$; in that case one might give the answer $\mathcal{L} = \{\epsilon\}$, but that's too "specialistic" and some books explicitly define alphabets as non-empty, as anyhow irrelevant.

(i) $\Sigma = \{a, b\}$ as an example for 2 or more symbols: That language is *not* regular. It would involve "*counting*" and in particular remembering the order in which way the a's and b's in the first half or a word ww in $\mathcal{L}$ are arranged, something which is not doable with finite memory.

(ii) $\Sigma = \{a\}$: The language represents words with an even number of a, which certainly can be represented by a regular expression:

$$(aa)^*$$

(ii) The automaton is not minimal. One can identify 1 and 3. That's the short answer (without goint through the split-algo).

(Continued on page 4.)

(iii) As usual, there is not one single possible solution. Here are a few, all start and end the same, of course. Only the middle part, the comment string itself, can be differently represented.

$$/\,o\ (o^*z\ \mid\ /)^*o^+\,/ \tag{1}$$
$$/\,o\ /^*(o^*z\,/^*)^*o^+\,/ \tag{2}$$
$$/\,o\ (o^*z\,/^*)^*/^*o^+\,/ \tag{3}$$
$$\tag{4}$$

The basic thing to avoid is to have a o immediately followed by a /, therefore we need a z in between. A more fine point is that there does not need to be a z at all, but still there may be a sequence of o's.

□

(Continued on page 5.)

Problem 2 Context-free languages and parsing (weight 26%)

2a LR(0)-DFA (weight 12%)

Consider the following context-free grammar:

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & S\mathbf{y} \mid \mathbf{x} \\ B & \rightarrow & \mathbf{y}S \mid y \end{array}$$

In the grammar, $\mathbf{x}$ and $\mathbf{y}$ are terminals, S, A, and B are non-terminals, with S as start symbol. After extending the grammar with a new non-terminal S' as new start-symbol and the corresponding production, do the following steps:

(i) give the *First*- and *Follow*-sets of the non-terminals.

(ii) give the DFA of LR(0)-items (numbering the states for later reference).

(iii) Is the grammar SLR(1) or not? Explain. In case the grammar is *not* SLR(1), identify corresponding conflicts in terms of in which state(s) they occur and what conflicting reactions occur under which input.

2b Bottom-up vs top-down (weight 7%)

Answer the following two questions, where you should try to keep the required examples simple. Note: it's not required to find the simplest possible examples, but please try not to use more than 3 non-terminals or more than 4 terminals (not counting **$**).

(i) Give an example of a context-free grammar which is LL(1) but *not* LR(0).

(ii) Give an example of a context-free grammar which is LR(0) but *not* LL(1).

Give a short explanation in each case, justifying why the chosen example does or does not belong to LL(1) resp. LR(0). It is not required to give parsing tables as justification.

2c Memory usage (weight 7%)

The following two grammars are SLR(1) (no proof or argument required for that), both representing the language $\mathbf{a}^*$:

(Continued on page 6.)

grammar G_1: $S \rightarrow A$, $A \rightarrow A\mathbf{a} \mid \boldsymbol{\epsilon}$ grammar G_2: $S \rightarrow A$, $A \rightarrow \mathbf{a}A \mid \boldsymbol{\epsilon}$

The task here is to compare the memory efficiency of the SLR(1) bottom-up parsers for the 2 grammars. When parsing $\mathbf{a}^n$ as input, what is the maximal stack size during the parser run. Use "big-O" notation, for instance using $\mathcal{O}(1)$ for *constant* stack memory usage, $\mathcal{O}(n)$ for stack-size linear in the size of the input string etc.).

You may use a small example runs as illustration of your argument. It's not required to give the SLR(1)-parsing table. □

Solution:

(i) The task is completely standard.

(i) The sets are given in Table 1 and the DFA is given in Figure 1.

Tabell 1: First- and follow-sets

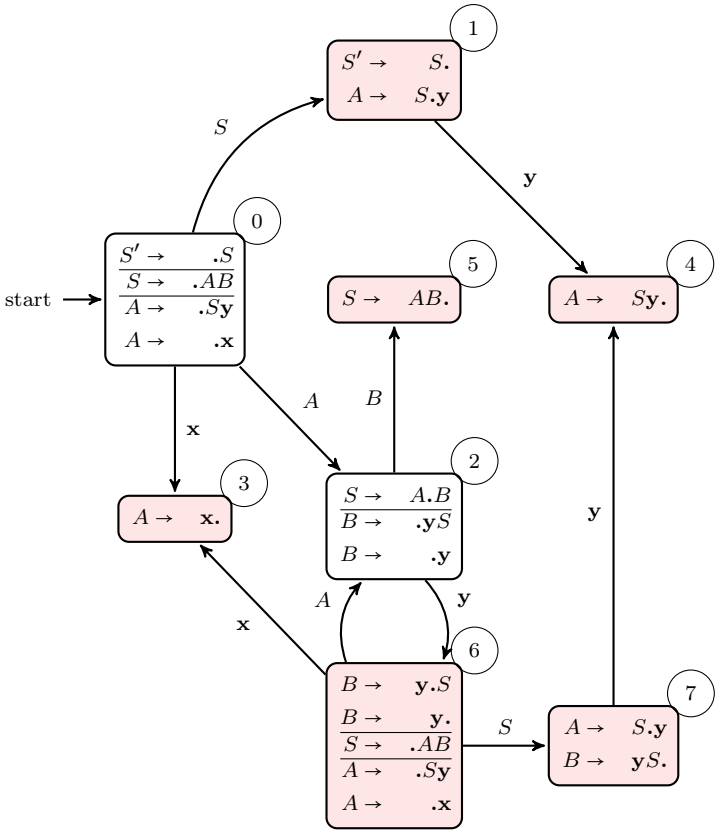
non-term.	*First*	*Follow*
S'	$\mathbf{x}$	$\mathbf{\$}$
S	$\mathbf{x}$	$\mathbf{\$}, \mathbf{y}$
A	$\mathbf{x}$	$\mathbf{y}$
B	$\mathbf{y}$	$\mathbf{\$}, \mathbf{y}$

(ii) The grammar is *not* SLR(1). That can be seen in state 7: Since *Follow*B contains $\mathbf{y}$ (a terminal which follows the "parser position $\mathbf{.}$" in one item), there's a shift-reduce conflict on symbol $\mathbf{y}$. Another "suspicious" state is 1, but this one is no conflict (as can be seen from the follow-set of S').

With the grammar changed with the additional production: state 6 is now also a state containing an complete item. That makes the state suspicious as well. The complete item is for a production with the left-hand side B. The follow of B does not contain x, so that one is fine, as well.

(ii) Unlike the previous one, this is about *grammars* not *languages*. It's best answered by remembering which features don't work for certain classes of *grammars* and quickly check if a simple example can be covered by the other class. Of course the grammars need to be *unambiguous*. So the task is to find a simple case of *unambigous grammars* building in the problematic productions for both LL(1) and LR(0).

(Continued on page 7.)



Figur 1: LR(0)-DFA

(i) *Problematic* for LR(0) are $\boldsymbol{\epsilon}$-productions. For those, a reduction-step is possible, and we need a state (resp. a corrsponding non-terminal) which allows also a shift. The following grammar is the simplest for that:

$$A \to \boldsymbol{\epsilon} \mid \mathbf{a} \tag{5}$$

For LL(1) parsing, $\boldsymbol{\epsilon}$-productions are unproblematic.[1] Note that the language is *finite* (i.e., not just regular, but basically *trivial* and it consists of only one symbol). It might sound unusual to use "recursive descent" in that situation, basically, there are only two cases to check: whether the next "symbol" is **$** or the next symbol is **a** followed by **$**. Still: technically, the grammar is not LL(1).

Alternatively, the following is a plausible simple solution, as well:

$$A \to \boldsymbol{\epsilon} \mid \mathbf{a}A \tag{6}$$

The grammar is *right-recursive* (which is fine for LL(1)) but the ϵ-production makes it non LR(0), as above.

Of course the *left-recursive* alternative

$$A \to \boldsymbol{\epsilon} \mid A\mathbf{a} \tag{7}$$

[1] Factually, transformations covered in the lecture to massage non-LL(1)-grammars in an equivalent representation which might become LL(1) (like left-factorization) routinely added $\boldsymbol{\epsilon}$ productions.

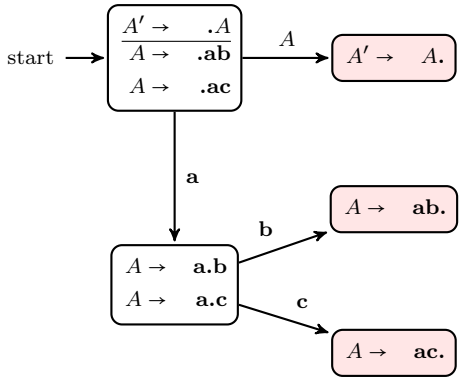
(Continued on page 8.)

for the same language would not be LL(1).

(ii) For the reverse-directions: LL(1)-parsers cannot deal with common left factors.

$$A \to ab \mid ac \tag{8}$$

Consequently, the grammar is not LL(1). It's LR(0) though: There is no reason for any conflict, as one can easily check. For reference, the corresponding LR(0)-DFA is given in Figure 2.



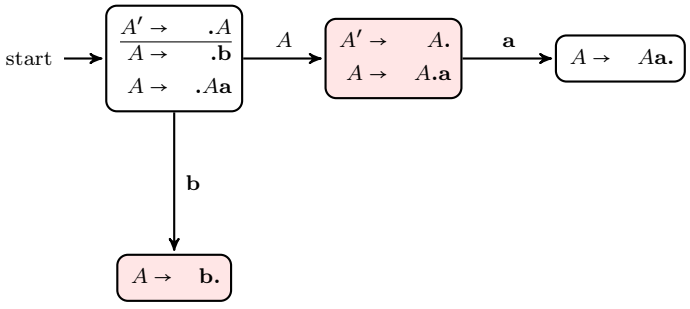
Figur 2: LR(0)-DFA

Remark: As mentioned, left-recursion is problematic for LL(1), as well. Thus, one might be tempted to use (7) as an example. It's certainly not LL(1) but unfortunately, the grammar is also *not* LR(1) (still containing an ϵ-production).

Additional remark: Even if we replaced the ϵ with a terminal **b** yielding

$$A \to \mathbf{b} \mid A\mathbf{a} \tag{9}$$

the grammar won't be LR(1):



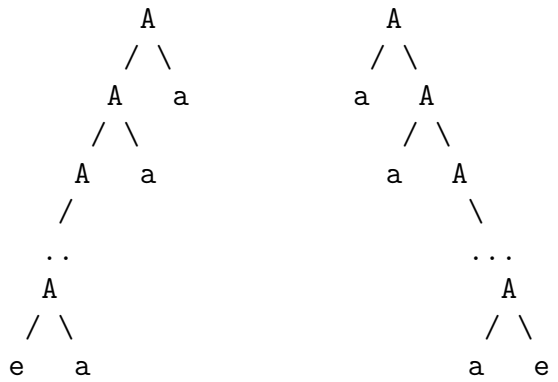
Figur 3: LR(0)-DFA ("**ba***")

(iii) The one on the *left* is $\mathcal{O}(1)$, the one on the right is $\mathcal{O}(n)$. One possible answer would of course be make the LR(0)-automaton again (which is simple enought) and take it from there. If one draws the automaton,

(Continued on page 9.)

one of them has a *loop* labelled **a** and the other not. The one with the loop (which is the consequence of the right-recursion $A \rightarrow \mathbf{a}A$) obviously shifts all the **a**'s, and that leads to $\mathcal{O}(n)$

It's not required here to give the full automaton here. Shorter answers along the lines "left-recursive rules doent not require to build up a stack, unlike right-recursive" are acceptable as correct as well, perhaps making use of the two different parse trees and how *bottom-up LR-parsers* treat them:



To build the right-hand tree bottom-up, one needs to remember a lot of **a**'s before one start's building the first finished tree, for the tree on the left, one can start right-away (all parsers work from left-to-right).

□

(Continued on page 10.)

Problem 3 Attribute grammars (weight 25%)

The lectures presented how to extract from three-address intermediate code a *flow graph*. The task here uses *a different approach!* Instead of taking three-address intermediate code as starting point, we use the *abstract syntax* and extract control flow information directly from there. We use *attribute grammars* for that.

We are dealing with a simple language, whose syntax is given by the grammar below. The form of non-terminals *assign* and *cond* are left undefined.

		productions	remarks
program	→	**begin** *stmt* **end**	**begin** and **end** carry a label
stmt	→	*stmt* **;** *stmt*	
	|	**while** *cond* **do** *stmt*	*cond* carries a label
	|	**if** *cond* **then** *stmt* **else** *stmt*	*cond* carries a label
	|	*assign*	*assign* carries a label

Contrary to the flow graphs presented in the lecture for three-address code, our "*abstract flow graphs*" consider each assignment and each condition as a separate node for the graph.

The task now is: add *semantic actions* to the grammar to calculate a control-flow information from a syntax tree. *Labels* are used to identify and represent nodes of our version of flow graphs.

Starting point: attribute `label` given We shall assume that the non-terminals *assign* and *cond* as well as the terminals **begin** and **end** all carry an an attribute `label`, containing a *label* value for indentification. These label values are *already filled in*. So you can make use of, for instance *assign*`.label` but you are not supposed to *set* the value. All label values are different.

Attributes `first` and `lasts` for *stmt*: Non-terminal *stmt* shall carry attributes `first` (containing a label) and `lasts` (containing sets of labels). They are supposed to contain the label of the condition/assignment executed *first*, respectively the *labels* of those executed *last*.

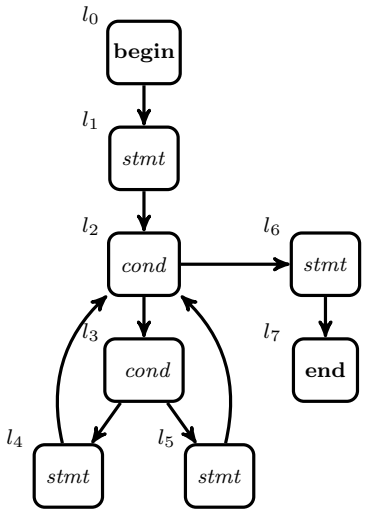
Attribute `succ`: Assume an attribute `succ` (containing a set of labels), intended to represent the *successor nodes* in terms of the control flow. In that way, they correspond to edges in the abstract flow graph.

For illustration: The left-hand side below contains a piece of concrete syntax where (for illustration) we have marked pieces with appropriate labels. A corresponding abstract flow graph is shown on the right-hand side. Note that

(Continued on page 11.)

after the evaluation of the attribute grammar, the `succ`-attributes indicate the succssor-nodes.

```
begin^l0
x := 9^l1 ;
while (x > 8)^l2
do { if      (y = 0)^l3
     then    x := 5^l4
     else    x := 6^l5
   } ;
x := 0^l6
end^l7
```



So: Give your answer in a filled out table of the following form. The semantic rules for the production *program* → **begin** *stmt* **end** are filled in already, making use of the notation $\{\ldots\}$ to represent sets.

	productions/grammar rules	semantic rules
0	*program* → **begin** *stmt* **end**	*stmt*.`succ` = {**end**.`label`} **begin**.`succ` = {*stmt*.`first`}
1	*stmt* → *assign*	
2	$stmt_0$ → $stmt_1$; $stmt_2$	
3	$stmt_0$ → **if** *cond* **then** $stmt_1$ **else** $stmt_2$	
4	$stmt_0$ → **while** *cond* **then** $stmt_1$	

□

Solution: The best way to attack (or present) the problem is to first do the two attributes `first` and `lasts`, and only afterwards, the successor. The first- and lasts-attributes are also easier, insofar they are synthesized, and for most people, purely synthesized attributes seem more natural. Therefore I start with those. The first- and lasts-attributes can be seen as *auxiliary*

(Continued on page 12.)

attributes used to enabling a more or less straightforward definition of the `succ`-attributes.

A good starting point is to fix, what *are* the actual attributes and for which nodes. In the text it is stated that *stmt* carries `first` and `lasts` (which is therefore required). It does not state that other terminals or non-terminals carry that; and they don't.

It is on the text not explicitly specified, which grammar symbols are supposed to carry `succ` as attribute. Indiractly in the graphical representation, it's indicated that *cond* and *stmt* carry that. What is not depicted in the picture are *assign*-non-terminals, [2] one has to figure out that also *those* are supposed to carry `succ` as attributes. Actually, in the concrete illustration, in the example code, the statements and the assignments are somehow "identical" in that thet "statements" are actually "assignments' (via the production $stmt \rightarrow assign$). One has to understand that also *assign* better carries an (inherited) attribute `succ`. If a otherwise correct solution stops determining the successors at *stmt* without inheriting it in a last step down to *assign*, is perhaps also acceptable, at least not too big an error. For *cond*, the graphics indicates that `succ` is a required attribute and the same for the "concrete syntax" code example.

An overview over the attributes and to which symbols they belong are shown in Table 2. The types of the attributes (one label resp. a set of labels) are given by the task and not repeated in the table. The attributes which are already given, namely `label`, are shown in parentheses. The ones in [brackets] are not actually needed, but they would not hurt either. The **end**-node should better not carry a `succ`-attribute (unlike **begin**), as there is no meaningful value to fill in. Practically, a realimplementation would leave a nil-pointer, but for the declarative framework of attribute grammars (where there are a priori no such notions as pointers), an attribute for wich there is not real definition is not adequate. Conceptually, the whole purpose of the labelled **end** node is to provide a successor label for those last statements of the "real" program (a "sentinel node"), to avoid having "nil-pointers" there. Therefore it's counter-productive to let **end** have an undefined/nil-pointer itself. One could accept a solution which adds `succ` to **end** and leave it undefined, even if it's not 100% kosher. Besides the already labelled symbols, *no* other grammar symbol should carry a label. It conceptually does not make sense; besides there's no mechanism to *add new* labels (and the text states all labels are supposed to be different).

Intuitively, the fact that the first and the last nodes/labels are synthesized may be seen from two facts: first the *leaves* of the syntax tree (assignments plus the special begin and end-nodes) are labelled already and thus in principle a statement which is an assignment carries the first- and lasts-information already (in the form of the label). Thus, the information can be

[2] Actually, since it's a fragment of a grammar, where *assign* and *cond* are left unspecified, those actually can be seen as playing the role of terminals.

(Continued on page 13.)

symbol	attributes
stmt	`first`, `lasts`, `succ`
assign	(`label`), [`first`, `lasts`], `succ`
cond	(`label`), [`first`, `lasts`], `succ`
begin	(`label`), `succ`
end	(`label`)
program	`first`, `lasts`

Tabell 2: Overview over attributes

propagated only "upwards" in the form of synthesized attributes. Secondly, the already filled in slot for the production for *program* makes it into a synthesized attribute. Of course, the pure fact that *program*.`first` is synthesized does not logically imply that `first` is synthesized for other grammar symbols, but is intended as inspiration.[3]

A final word on why first and last nodes are synthesized. We argued that that leaves of the tree, the "base cases", carry that information already filled in. What makes it a bit strange is that *cond* carries a label as well despite the fact that *cond* nodes in a syntax tree are *not* leaves. Here one has to understand the role of `first` and `lasts`. In principle *cond* is not supposed to carry those attributes resp it's not necessary/required (that's why it's in brackets in the table). But one can can come up with a reasonable solution where *assign* and *cond* also carry the attributes `first` and `lasts`. For *assign*, it's pretty obvious how to define that, for *cond*, the only meaningful definition is that the firsts and lasts of *cond* corresponds to the firsts and lasts of the statement it belongs to ($stmt_0$ in the grammar). It's omitted in the given solution.

Attributes `first` and `lasts` So, let's start then with Figure 4. Clearly, the semantics rules are all bottom-up. It's basically a recursive definition of the first "node" and the set of last "nodes", represented by the labels.

One could accept if the non-terminal *program* were not labelled insofar the task/table may seem to imply that for that production it's done already and that it's not really needed for the `succ`-label anyway. Note also that the definition does not refer to `succ` at all; as said, the first- and lasts-attributes are independent from the definition of the successor.

Attribute `succ` Now, given the labels for the first and the last nodes, the rules for `succ` are shown in Table 5. Now the perspective changes:

[3] In the lecture, there had been examples where attributes of the same name had been synthesized for one symbol/node class, but inherited for another (for instance for types). Here, it's simpler. Of course, one could in general always avoid that situation by simply using two different attribute names. On the other hand, that may be confusing as well, as really it's a "type" which is synthesized a one symbol but inherited at another.

(Continued on page 14.)

	productions/grammar rules	semantic rules
0	*program* → **begin** *stmt* **end**	[*program*.`first` = **begin**.`label`] [*program*.`lasts` = {**end**.`label`}]
1	*stmt* → *assign*	*stmt*.`first` = *assign*.`label` *stmt*.`lasts` = {*assign*.`label`}
2	$stmt_0$ → $stmt_1$; $stmt_2$	$stmt_0$.`first` = $stmt_1$.`first` $stmt_0$.`lasts` = $stmt_2$.`lasts`
3	$stmt_0$ → **if** *cond* **then** $stmt_1$ **else** $stmt_2$	$stmt_0$.`first` = *cond*.`label` $stmt_0$.`lasts` = $stmt_1$.`lasts` ∪ $stmt_2$.`lasts`
4	$stmt_0$ → **while** *cond* **then** $stmt_1$	$stmt_0$.`first` = *cond*.`label` $stmt_0$.`lasts` = {*cond*.`label`}

Figur 4: AGrammar for `first` and `lasts`

it's no longer strictly *synthesized*, That can already be seen in the slot for *program* which has been filled out already. The core intuition is: the statement representing the program as such (i.e., the *stmt* mentioned in the filled-out production for *program*) has its successor filled out by the corresponding semantic rule (the slot for rule 0). Now, this information has to be *pushed down* the syntax tree.

(Continued on page 15.)

	productions/grammar rules	semantic rules
0	*program* → **begin** *stmt* **end**	*stmt*.succ = {**end**.label} **begin**.succ = {*stmt*.first}
1	*stmt* → *assign*	*assign*.succ = *stmt*.succ
2	$stmt_0$ → $stmt_1$; $stmt_2$	$stmt_1$.succ = {$stmt_2$.first} $stmt_2$.succ = $stmt_0$.succ
3	$stmt_0$ → **if** *cond* **then** $stmt_1$ **else** $stmt_2$	*cond*.succ = $stmt_1$.first ∪ $stmt_2$.first $stmt_1$.succ = $stmt_0$.succ $stmt_2$.succ = $stmt_0$.succ
4	$stmt_0$ → **while** *cond* **then** $stmt_1$	*cond*.succ = {$stmt_1$.first} $stmt_1$.succ = {*cond*.label}

Figur 5: AGrammar for succ

Problem 4 Code generation (weight 18%)

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*).

4a Register descriptors (weight 5%)

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.

4b Local optimization (weight 13%)

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.

(Continued on page 16.)

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

> 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.

(Continued on page 17.)

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 1 and 2 makes use of that.

Listing 1: 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 2: 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 ------------------------------------------------------------------
```

□