Universitetet i Oslo
Institutt for Informatikk

Reliable Systems

Martin Steffen

# INF 5110: Compiler construction

Spring 2018

# Collection of older exam questions

14. 05. 2018

**(including hints for solutions)**

**Issued: 14. 05. 2018**

## Contents

**Abstract**

This is a collection of exams from earlier years. They are not the originals but translated to English (but I more or less tried to keep true to the formulations). Additionally, there are hints for solutions, (made available later) also taken from those earlier exams.

In the solutions as I have written down here, there is often *more* text than what is expected when answering an exam, such as explaining what is generally expected in such a question, about the background,[1] or how to approach it. In contrast, in an exam, one is very much encouraged to keep explanations more to the point of the actual question at hand.

*Disclaimer:* Care has been taken to keep it error-free here; I do *not,* however, give guarantees for 100% correctness, and an error here can not be taken as argument when defending own errors.

Also: it's unclear whether throughout the years, exactly the same *pensum* was required. The pensum of 2016, 2017, and this semester 2018, corresponds roughly (but not 100%) to the one from 2015, but I have no overview over earlier semesters. Thus, earlier exams may cover more/different material or left out some material, which has been added to the pensum later on. The text here is just a "matter-of-fact" repository of earlier exams (and not all of them have been included yet). Peruse at your leisure.

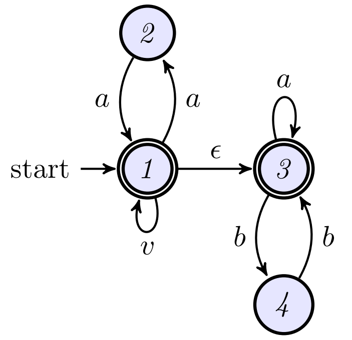
# 1 2005

### Exercise 1 (Regular expressions and automata (0%))

(a) Use Thompson's construction to construct a NFA for the following regular expression

$$(aa \mid b)^*(a \mid cc)^*$$

(b) Write the following NFA as *regular expression.*




(c) Turn the NFA from the previous sub-problem into a *DFA*.

**Solution:**

(a)

□

[1]Especially in the footnotes.

**Exercise 2 (context-free grammars and parsing (0%))**

Consider the following grammar $G_1$:

$$\begin{array}{rcl} E & \rightarrow & S\ E \mid \mathbf{num} \\ & \mid & \\ S & \rightarrow & -\ S \mid +S \mid \epsilon \end{array}$$

$E$ and $S$ are non-terminals, +, -, and **num** are terminals (with the usual interpretation). The start symbol is $E$ (not $S$).

(a) Describe short how sentences generated by $G_1$ look like, and give one example of a sentence consisting of 4 terminal symbols

(b) Give a *regular expression* representing the same sentences as $G_1$.

(c) Give a short argument determining which of the following 5 groups the the *grammar* belongs to (more than one may apply):

(i) LR(0)
(ii) SLR(1)
(iii) LALR(1)
(iv) LR(1)
(v) none of the above.

Consider next a different grammar $G_2$:

$$F \rightarrow +\ F \mid -\ F \mid \mathbf{num}$$

Here, $F$ is a non-terminal (and, obviously, the start symbol). The terminals are unchanged: +, -, and **num**

(d) Give a LR(0)-DFA for $G_2$, where the grammar has been extended by a new producion $F' \rightarrow F$ and where $F'$ is taken as the start symbol of the extended grammar. Give a number to each state of your DFA for identification.

(e) Given the DFA thus constructed: which type(s) of grammar is $G_2$, again with a short explanation. (Cf. question (c) from above for the classification).

(f) Give the *parsing table* for $G_2$, fitting to the type of grammar

(g) How will the sentence following sentence be parsed

$$--\mathbf{9}$$

Give your answer by showing the *stack-content* and *input* (as done in the book) for each of the shift- or reduce-steps done while parsing the sentence. □

**Solution:**

(a)

(b)

(c) `-5` an be derived as follows

```
          S
        /   \
       /     \
     S         E
    / \       / \
   -   S     S   E
       |     |   |
       e     e   5
```

```
          S
        /   \
       /     \
      -        E
              / \
             S    E
            / \   |
           -   S  5
               |
               e
```

where e stands fro the empty word. So that means the grammar does not fall in any of the given categories.

(d)

(e) seems to me the automaton shows it's LR(0). But check again.

□

**Exercise 3 (Attribute grammars and type checking (0%))**

(a) The following is a (fragment of a) grammar for a language with *classes.*

$$
\begin{array}{rcl}
\mathit{class} & \rightarrow & \textbf{class name } \mathit{superclass}\,\{\,\mathit{decls}\,\} \\
\mathit{decls} & \rightarrow & \mathit{decls}\,\textbf{;}\,\mathit{decl} \mid \mathit{decl} \\
\mathit{decl} & \rightarrow & \mathit{variable\text{-}decl} \\
\mathit{decl} & \rightarrow & \mathit{method\text{-}decl} \\
\mathit{method\text{-}decl} & \rightarrow & \mathit{type}\ \textbf{name}\,\textbf{(}\,\mathit{params}\,\textbf{)}\,\mathit{body} \\
\mathit{type} & \rightarrow & \textbf{int} \mid \textbf{bool} \mid \textbf{void} \\
\mathit{superclass} & \rightarrow & \textbf{name}
\end{array}
$$

Words in *italics* are meta-symbols, words or symbols in **boldface** are terminal symbols (and **name** represents a name the scanner hands over. You can assume that **name** has an attribute `name`.

Methods with the same name as the class are *constructors*, and, as a rule, constructors must have the type `void`.

The task now is: formulate semantic rules for each production in the following *fragment* of an attribute grammar. Start by deciding which *attributes* you need.

Hint: the solution does not require a symbol table.

| | productions/grammar rules | | | semantic rules |
|---|---|---|---|---|
| 1 | *class* | → | **class name** *superclass* **{** *decls* **}** | |
| 2 | *decls* | → | *decls* **;** *decl* | |
| 3 | *decls* | → | *decl* | |
| 4 | *decl* | → | *variable-decl* | Not to be filled out |
| 5 | *decl* | → | *method-decl* | |
| 6 | *method-decl* | → | *type* **name (** *params* **)** *body* | |
| 7 | *type* | → | **int** | |
| 8 | *type* | → | **bool** | |
| 9 | *type* | → | **void** | |
| 10 | *superclass* | → | **name** | |

(b) Assume we are dealing with a language with classes and subclasses. All methods are *virtual* (such that they can be overwritten). Assume the following class definitions:

```
class A {
  int i;
  void P { ... AP ... };
  void Q { ... AQ ... };
}

class B extends A {
  int j;
  void Q { ... BQ ...};
  void R { ... BR ...};
}

class C1 extends B {
  void P { ... C1P ...} ;
  void S { ... C1S ...} ;
}

class C2 extends B {
  int k
  void R { ... C2R ...} ;
  void T { ... C2T ...} ;
}
```

Show how objects of classes $C_1$ and $C_2$ are structure (show their layout) and draw the virtual function table[2] for each of the classes. Use the "names" shown in the above method bodies to indicate elements in the virtual function tables.

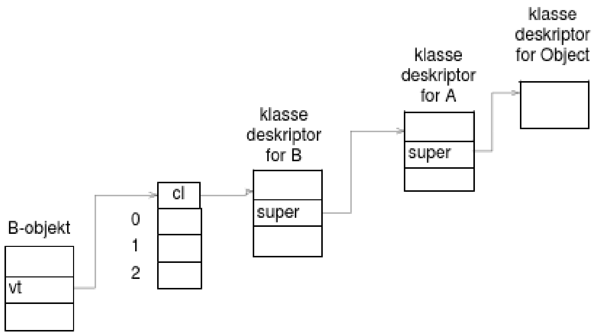
(c) We introduce an `instanceof` operator as in Java. The boolean expression

*refExpr* **instanceof** *class*

is "true" if the object pointed at by *refExpr* is of a class which is not "null", and which is class *class* or a subclass of *class*. Otherwise, the value of the expression is "false".

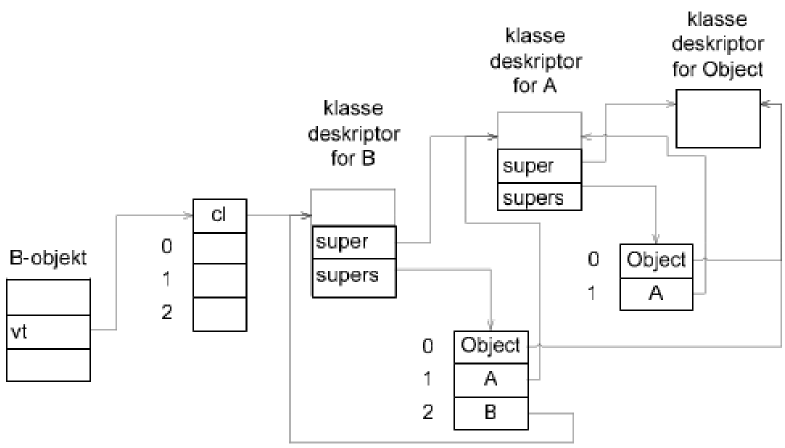
[2] name

To implement this operation, we extend the virtual function table with a pointer to class descriptors; there is one class descriptor for each class in the program. Each class descriptor contain a variable "super" pointing to the class descriptor of its superclass. Classes without an explictly given superclass have the specific class **Object** as superclass. The example figure below illustrates the concept for an object of class $B$.




Sketch an algorithm which calculates the value of *refExpr* **instanceof** *class*

(d) To make the test of **instanceof** more efficient and inspired by the concept of *display*/context vector for nested blocks, we instroduce a table "*super*" which, for a given class, contains all superclasses including the class itself. This table uses as index the "subclass-level", with 0 for **Object**, with 1 for the programs root class, etc. In our example, class $A$ has level 1, $B$ has 2, and $C_1$ and $C_3$ both level 3. In our example, the class descriptors which includes the "*super*"-tables look as follows:




*Explain* how this representation can make the implementation of the **instanceof**-operator. To illustrate that, we introduce two more classes:

```
class C11 extends C1{...}
class C21 extends C1{...}
```

Give the class descriptors for those two new classes $C_{11}$ and $C_{21}$ and show how the following tests are done.

```
rC11 = new C11();
rC11  instanceof C1;  // (1)
rC11  instanceof C2;  // (2)
```

□

**Solution:** □

# 2 2006

**Exercise 4 (Parameter passing and attribute grammars (0%))**

The following is a fragment of a grammar for a language with procedures (uninteresting parts are omitted for the current problem set). All procedures have *one* parameter that this parameter is either *"by-value"*, *"by-reference"*.(indicated by they keyword **ref**), or *"by-value-result"* (indicated by the keyword **result**).

$$
\begin{array}{rcl}
\textit{procedure} & \rightarrow & \textbf{proc id (}\ \textit{param}\ \textbf{)}\ \textit{stmt} \\
\textit{param} & \rightarrow & \textit{type}\ \textbf{id} \mid \textbf{ref}\ \textit{type}\ \textbf{id} \mid \textbf{result}\ \textit{type}\ \textbf{id} \\
\textit{call} & \rightarrow & \textbf{id (}\ \textit{exp}\ \textbf{)} \\
\textit{exp} & \rightarrow & \textbf{id} \\
\textit{exp} & \rightarrow & \textbf{id [}\ \textit{exp}\ \textbf{]} \\
\textit{exp} & \rightarrow & \textit{exp}\ \textit{aritop}\ \textit{exp}
\end{array}
$$

The following 2 programs declare a variable `i` and a procedure `change`; afterwards, 1 is assigned to `i`, the procedure is called with $i$ as argument and finally prints the content of `i`. The difference between the first and the second version of the program is the parameter-passing mode: the first uses call-by-reference, the second call-by-value-result. We assume standard scoping rules apply.

```
{
  int i;
  proc change (ref int p) {
    p = 2; i = 0;
  };
  i = 1;
  change(i);
  write(i);
}
```

```
{
  int i;
  proc change (result int p) {
    p = 2; i = 0;
  };
  i = 1;
  change(i);
  write(i);
}
```

(a) Assume that the semantics for "call-by-value-result" is such that the address (location) of the actual parameter is determined at the time of the procedure *call* (procedure entry).

What is the output of program 1 and program 2 upon execution?

(b) Assume that the semantics for "call-by-value-result" is such that the address (location) of the actual parameter is determined a the time of the procedure *return* (procedure exit).

(c) The easy rule governing procedure calls in this language "by-reference" or "by-value-result" is as follows: such procedures can be called only where the expression is either a simple variable (**id**) or an indexed variable (**id [** *exp* **]**).

Fill out the missing entries in the following attribute grammars such that the attribute `ok` for *call* is true the call is done following the given rule and false, otherwise.

The symbol table is set up targeted towards this language rule such that the names of procedures are associated with a value which indicates whether the given procedure uses its parameter "by-value", "by-reference", or "by-value-result" (with values *value*, *ref*, or *result*, respectively). A call *lookupkind*(**id**.`name`) gives in which way the procedure with the name **id**.`name` is defined.

It's not required here to check whether the procedure name **id** in a call-expression is actually declared.

| productions/grammar rules | | | semantic rules |
|---|---|---|---|
| *procedure* | $\rightarrow$ | **proc id (** *param* **)** *stmt* | *insert*(**id**.`name`, *param* .`kind`) |
| *param* | $\rightarrow$ | *type* **id** | |
| *param* | $\rightarrow$ | **ref** *type* **id** | |
| *param* | $\rightarrow$ | **result** *type* **id** | |
| *call* | $\rightarrow$ | **id (** *exp* **)** | *call* .`ok` = |
| *exp* | $\rightarrow$ | **id** | |
| $exp_1$ | $\rightarrow$ | **id [** $exp_2$ **]** | |
| $exp_1$ | $\rightarrow$ | $exp_2$ *aritop* $exp_3$ | |

□

**Solution:** □

### Exercise 5 (Context-free grammars and parsing (0%))

Consider the following grammar $G$. In the grammar, $S$ and $T$ are nonterminals, # and **a** are terminals, and $S$ is the start symbol.

$$\begin{array}{rcl} S & \rightarrow & T\,S \\ S & \rightarrow & T \\ T & \rightarrow & \#\,T \\ T & \rightarrow & \mathbf{a} \end{array}$$

(a) Determine the *First*- and *Follow*-sets for $S$ and $T$. Use **\$**, as usual, to represent the "end-of-file".

(b) Forumalate in your own words which sequences of terminal symbols are generated starting from $S$.

(c) Is it possible to represent the language of $G$ (consisting of # and **a** symbols) by a *regular expression*. Explain, if the answer is "no", resp. give a corresponding regular expression if the answer is "yes".

(d) Introduce a new start symbol $S'$ with a production $S' \rightarrow S$. Give the $LR(0)$-DFA for $G$ right for that grammar. Give numbers to the states of the DFA.

(e) Give a short argument determining which of the following 5 groups the grammar belongs to; more than one answer is possible:

(i) LR(1)
(ii) LALR(1)
(iii) SLR(1)
(iv) LR(0)
(v) none of the above

Hint: determine possible conflicts in the constructed DFA and/or if the grammar is unambiguous.

(f) Give the *parsing table* for $G$, fitting the grammar type.

(g) Show how the sentence "**a # a**" is being parsed. Do that, as done in the book, by writing the stack-contents and input for each shift- or reduce-operation executed during the parsing. Indicate also the numbers of the states on the stack (as in the book). □

**Solution:**

```
I cannot really draw the automaton in an email, but here's how I think
it goes. I have not checked too cearefully, so errors might be in
there. Anyhow, here's my attempt




----------------
0: S' -> S
   S  -> . T S
   T  -> . # T
   T  -> . a
----------------


----------------
1: S -> T . S
   S -> . T S
   S -> . T
   T -> . # T
   T -> . a
----------------


-------------------
2: S -> T S .
-------------------


-----------------
3: S -> T.
   T -> T . S
   S -> . T S
   S -> . T
   T -> . # T
   T -> . a
-----------------
```

```
------------------
4:  T -> # . T
    T -> . # T
    T -> . a
------------------

------------------
5: T -> a.
-----------------

-----------------
6: S -> T . S
-  S -> . T S
   S -> . T
   T -> . # T
   T -> . a
-----------------


============= EDGES ========================


[0]  --T--> [1]
[0]  --S--> [2]
[0]  --#--> [4]
[0]  --a--> [5]

[1] --S--> [2]
[1] --T--> [3]
[1] --#--> [4]
[1] --a--> [5]


[2] : no outgoing edges

[3] --a--> [5]
[3] --#--> [4]
[3] --S--> [2]
[3] --T--> [6]




[4] --#--> [4]
[4] --a--> [5]


[5]: no outgoing edge



[6] --S--> [2]
```

```
[6] --T--> [3]
[6] --#--> [4]
[6] --a--> [5]
------------------------------------------


Now, Follow-sets (the first-sets are not so important)


  Follows

S'   $
S    $
T    $ , #

----------------------

Now for the conflicts. Suspicious are stats only 3.



The other 2 states with complete items are harmless.



So we have to look at the follow sets., but S has not # in its follow
set. Therefore it's fine for SLR.
```

□

**Exercise 6 (Classes and virtual tables (0%))**

(a) Assume a language with classes and subclasses. *All* methods are virtual, such that they can be redefined in subclasses.

The class `Graph`, together with classes `Node` and `Edge`, defines graphs, which consist of `Node`-objects which are connected via `Edge`-objected. An instance of class `Graph` represents graphs. All nodes of the graph are assumed to be reachable from a node represented by the attribute `startNode`, which contains a references to a `Node`-object.

Parts of the class definitions irrelevant for the problem are indicated by "...".

```
class Node { ... }
class Edge { ... }

class Graph {
  Node startNode;
  void connect(Node n1, n2) {
    ... // connects two Nodes by creating an Edge-object ...
  };
}
```

The following classes define *subclasses* (`City` and `Road`) of `Node` and `Edge`, respectively. Furthermore given is a subclass `RoadAndCityGraph` of `Graph`, and a subclass `TravelingSalesmanGraph` of `RoadAndCityGraph`. The method `display` will draw the graph with `startNode` as starting point.

```
class City extends Node {
    String name;
    ...
}

class Road extends Edge {
    String name;
    int distance;
    ...
}


class RoadAndCityGraph extends Graph {
    String country;
    void connect(Node n1, n2) {
        ... // connects to city objects treates as Nodes,
            // by creating a Road object
    };
    void display () {
        ... // display Roads and City with names
    }
}

class TravelingSalesmanGraph extends RoadAndCityGraph {
    void display () {
        ... // display cities with names and roads
            // with name and distance
    };
}
```

Show how objects of the classes `Graph`, `RoadAndCityGraph`, and `TravelingSalesmanGraph` are structured (show their layout) and draw the *virtual table* for each of the objects. Use names of the form ⟨*classname*⟩ :: ⟨*methodname*⟩ to indicate which definition is associated with each object.

(b) Assume that classes `Node` and `Edge` are defines as *inner classes* of `Graph` and furthermore that inner classes can be *redefined* in subclasses in the same way that virtual methods can. One may well speak of *virtual classes* then. Redefined classes *automatically* become subclasses for the corresponding virtual classes. For example, class `RoadAndCityGraph` is a subclass of class `Node` in `Graph`.

```
class Node { ... }
class Edge { ... }

class Graph {
  Node startNode;
  void connect(Node n1, n2) {
    ... // connects two Nodes by creating an Edge-object ...
  };
}



class RoadAndCityGraph extends Graph {
    class City {
        String name;
        ...
    }

    class Road {
        String name;
        int distance;
        ...
    }

    String country;
```

```
    void connect(Node n1, n2) {
        ... // connects to city objects treates as Nodes,
            // by creating a Road object
    };
    void display () {
        ... // display Roads and City with names
    }
}
class TravelingSalesmanGraph extends RoadAndCityGraph {
    void display () {
        ... // display cities with names and roads
            // with name and distance
    };
}
```

In the same way as the virtual table for virtual methods is used when calling a virtual method, we now also make use of an *additional virtual table* for the instantiation of objects from virtual classes. For instance, the method *connect* of class `Graph` contains code to generate a new `Edge`-object. If this method therefore is called on a `RoadAndCityGraph`-object, it is supposed to generate an `Edge`-object as it is defined in class `RoadAndCityGraph`.

Show how such a virtual table for virtual classes can look like. Don't include in the representation the virtual table from subproblem (a).

Explain how this new virtual table is used when executing `new Edge()` in method `connect` in the class `Graph`. □

# 3 2007

**Exercise 7 (Code generation (-%))**

(a) Given is the program from Listing 1. 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 1: 3-address code example

```
a := input
b := input
d := a + b
c := a * b    // <- looky here
if ( b < 5) {
  while (b < 0 ) {
    a := b + 2
    b := b + 1
  }
  d := 2 * b
} else {
  d := b * 3
  a := d - b
}
output a
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 live-ness algo. But actually, the program in simple enough so one might even more easily just look at the program and figure out by "carefully thinking" which of the variables at the specific line are live and which are not. Note: it's *not* required to give the values for the *inLive* and *outLive* points throughout the CFG. Other exam questions *do* require the full construction (partition the intermediate code, show the CFG, and show the liveness result for all positions in the graph), but here one is allowed to simply give the result (it's easy enough).

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

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

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

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

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

# 4 2009

### Exercise 8 (Code generation (%))

Consider the following program in 3-address intermediate code.

Listing 2: 3-address code example

```
1  a := input
2  b := input
3  t1 := a + b   // line 3
4  t2 := a * 2
5  c := t1 +  t2
6  if a < c goto 8
7  t2 := a + b
8  b := 25      // line 8
9  c := b + c
10 d := a - b
11 if t2 = 0 goto 17
12 d := a + b
13 t1 := b - c
```

```
14 c := d - t1
15 if c < d goto 3
16 c := a + b
17 output c      // line 17
18 output d
```

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

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

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

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

Assume that all variables are dead after the last instruction.

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

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

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

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

**Solution:**

(a) The basic blocks are indicated as comments in the code. The line numbers shift therefore, of course.[3] The first line indicates a basic block, targets of (conditional) jumps indicated basica blocks, and lines after (conditional) jumps indicate basic blocks.

Listing 3: 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
```

[3]Note that Louden favors a 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.

```
t1 := b - c
c := d - t1
if c < d goto 3
---------- B6 ------------------
c := a + b
----------- B7 ------------------
output c    // line 17
output d
--------------------------------------
```

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

(c) A possible rule could be

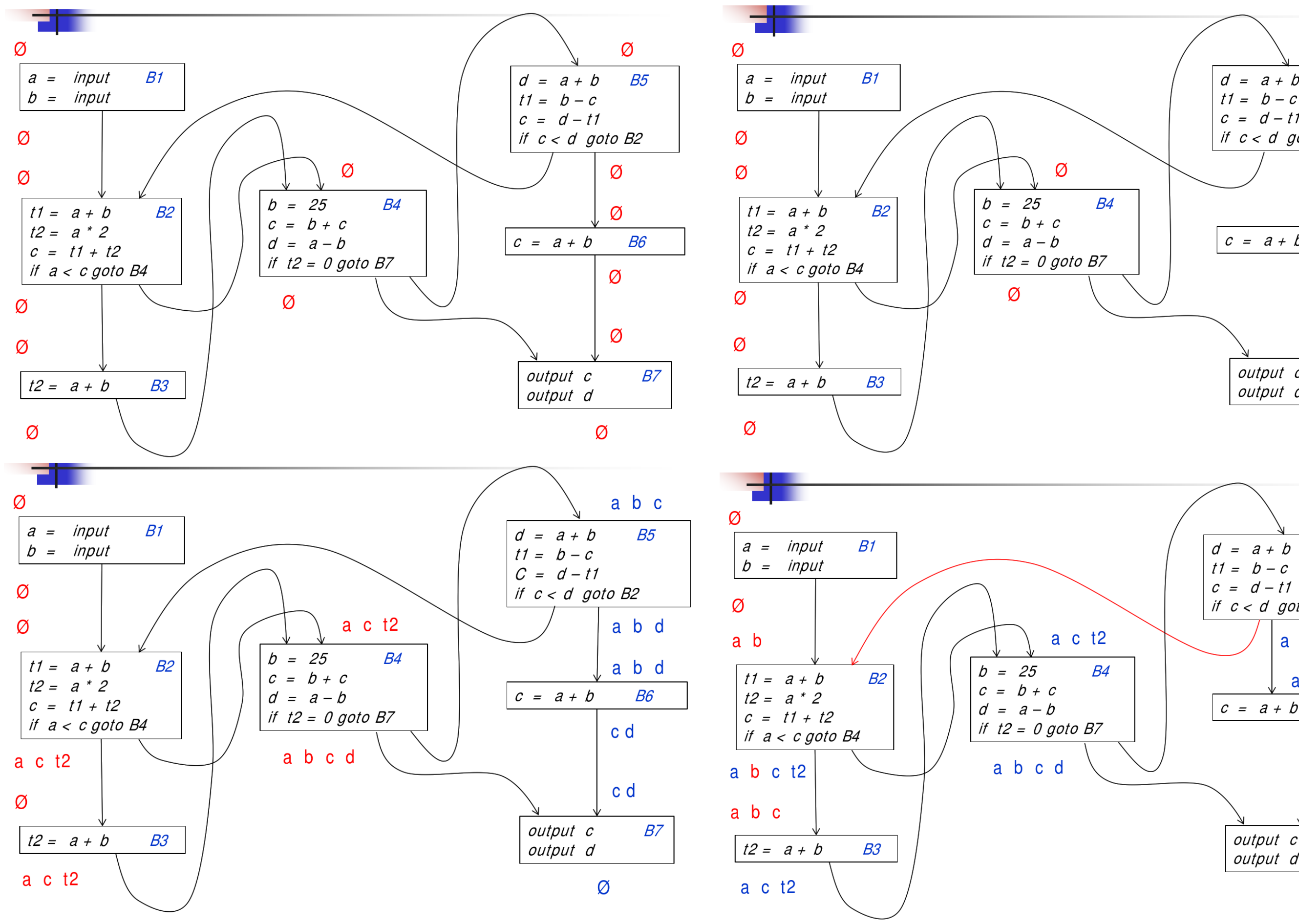
> All temporaries which are *used* in a given basic block must be assigned to ("defined") in the same before the (first) use.

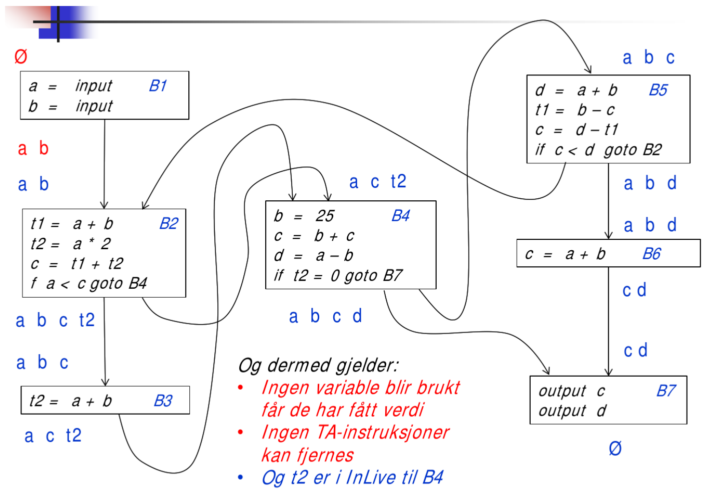
Another way of saying it is:

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

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

(e) Liveness:






## 5 2010

### Exercise 9 (Code generation (–%))

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

```
t1 := a − b
t2 := b − c
```

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

Listing 4: 2AC

```
MOV   a, R0
MOV   b, R1
SUB  R1, R0
SUB   c, R1
```

Ellen disagrees. Who is right? Explain your answer.

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

So: an "easy" way to see that the code generation in the book won't generate the code of Listing 4 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 5: 2AC (bonus)

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

□

## 6 2011

**Exercise 10 (CFGs and Parsing (25%))**

Given are the following *3 separate grammars:*

$$A \rightarrow \mathbf{b}A\mathbf{c} \mid \epsilon \quad (1)$$

$$A \rightarrow \mathbf{b}A\mathbf{b} \mid \mathbf{b} \quad (2)$$

$$A \rightarrow \mathbf{b}A\mathbf{b} \mid \mathbf{c} \quad (3)$$

Symbol $A$ is the start symbol and the (only) non-terminal, and **b** and **c** are terminals.

(a) For all three grammars:

(i) Calculate the *First*- and *Follow*-sets of $A$.

(ii) After extending the grammar with a new start symbol and production $A' \rightarrow A$, draw the LR(0)-DFA.

(iii) Which of the 3 grammars is SLR, if any? Do the same for LR(0).

(b) For each of the 3 grammars: is the grammar LR(1)? It's possible to determine and explain that without referring to the LR(1)-DFA, but it's ok to draw the LR(1) first and use it for the answer.

(c) Which of the languages generated by the grammars is *regular?* In case of a "yes", give a regular expression capturing the language of the respective grammar. In case of a "no" answer: give a short explanation.

(d) Draw a *parsing table* for grammar (1) and take care that it's *free from conflicts.* Give a step-by-step LR-analysis of the sentence "**bbcc**" in the same way as done in [Louden, 1997, page 213, Table 5.8]

**Solution:**

(a)

(i) The *First* and *Follow* sets are as follows:

| Nr. | *First*($A$) | *Follow*($A$) |
|---|---|---|
| 1) | $\{\mathbf{b}, \epsilon\}$ | $\{\mathbf{c}, \$\}$ |
| 2) | $\{\mathbf{b}\}$ | $\{\$, \mathbf{b}\}$ |
| 3) | $\{\mathbf{b}, \mathbf{c}\}$ | $\{\$, \mathbf{b}\}$ |

(ii) We give directly the DFAs, not going through the NFA's as intermediate step. In the direct construction (which may be faster than the indirect way over the NFAs), the core is to build the *closure* correctly. States containing *complete items* are shown shaded slightly in red. This is not part of the task, just done for illustration, as in the slides of the lecture.

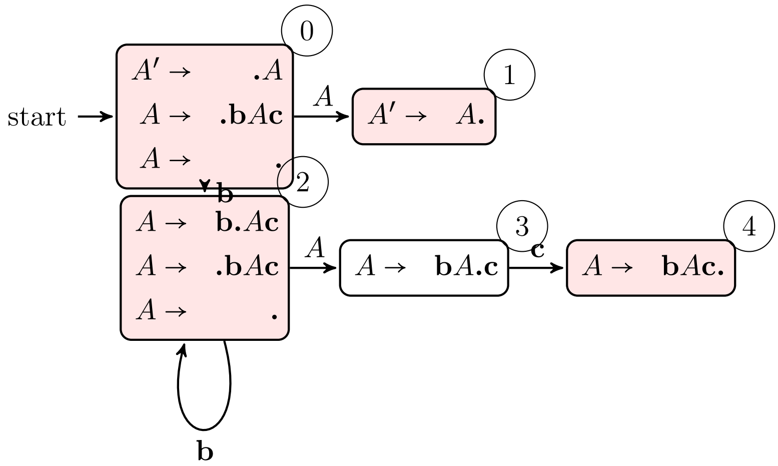



Figure 1: DFA for $A \to \mathbf{b}A\mathbf{c} \mid \epsilon$

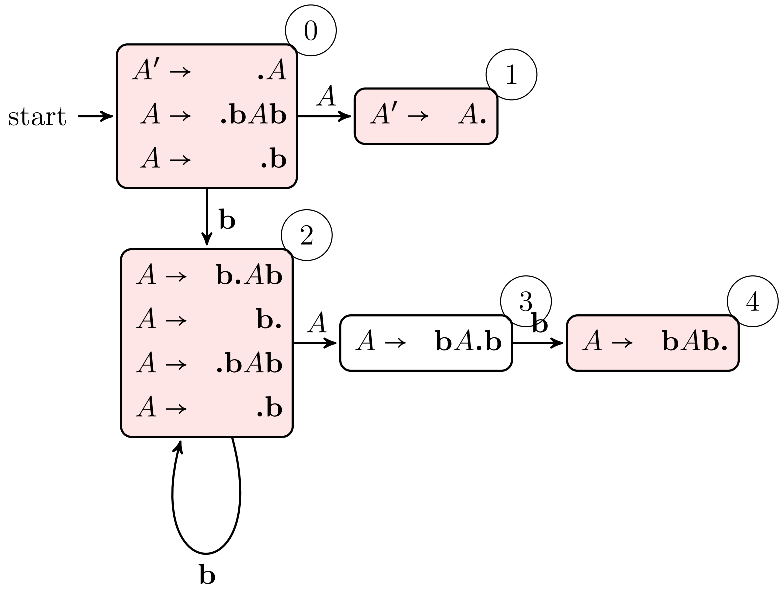



Figure 2: DFA for $A \to \mathbf{b}A\mathbf{b} \mid \mathbf{b}$

(iii) A classification of the different grammars is given in overview in Table 1 (covering also other subproblems). To determine whether the grammar is LR(0), resp. SLR, or not: For each of the DFA's, one has to look for *conflicts.*

i. For LR(0): this question is to be answered without looking into the follow-sets. Of course, when by looking into the follow-sets and finding out that an LR(0)-DFA is *no* SLR, then it's clear that it's also not LR(0). Anyhow, the third automaton has *no* LR(0)-conflicts: in each state it is *either* a shift possible *or else* a reduce, but not both at the same time. Reduce-steps are doable in states containing *complete* items (here marked reddish). It's immediate that in the automaton for $G_3$, there

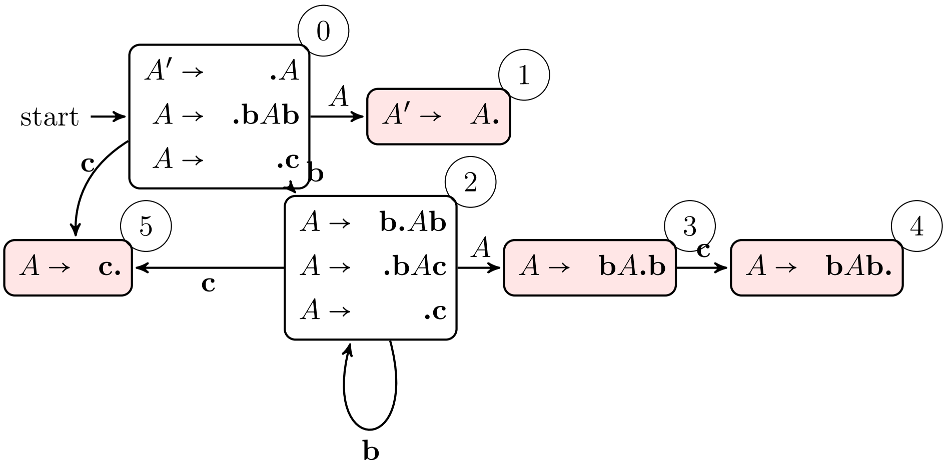



Figure 3: DFA for $A \to \mathbf{b}A\mathbf{b} \mid \mathbf{c}$

| grammar | regular | LR(0) | SLR(1) | LR(1) |
|---|---|---|---|---|
| $G_1$ | no | no | yes | [yes] |
| $G_2$ | yes | [no] | no | no |
| $G_3$ | no | yes | [yes] | [yes] |

Table 1: Classification (overview)

are no outgoing edges in the states containing a complete item (= states where a reduce is possibe). Thus, there is no shift-reduce conflict and thus the grammar is LR(0). It's equally trivial to see that this criterion is violated in the first two automata, they do have shift-reduce conflicts.

ii. For *SLR:* For $G_3$, the answer follows from the fact that the grammar is LR(0) already. For the others, we need to look at the follow sets, especially for the "suspicious" states, where there's an LR(0)-conflict. This time we need to consult the follow sets to see if or if not they disambiguate the situation.
In $G_1$, we need to check states 0 and 2 specifically (the other states either contain no complete item or contain *only one* complete item and no non-complete items). In both cases, we have to check for shift/reduce conflicts (that there is no reduce-reduce conflict, is clear; there is only one complete item in each state). For both states, we need to check the terminal **b**. Since it's not contained in the *Follow*-set of $A$, the grammar $G_1$ is SLR.
For $G_2$, state 2 is suspicious. In this case, the relevant terminal **b** is contained in the follow-set of $A$, hence $G_2$ is *not* SLR.

(b) Concerning LR(1): $G_1$ and $G_3$ are immediately clear, as they are already SLR.

Remains $G_2$. This grammar is *not* LR(1). The grammar is *unambiguous*, so we have to do it another way: When parsing a string of **b**'s of arbitrary length, there will be a point "in the middle" where the parser needs to decide, that, from now on, it's the second half (in the first half, the parser may push onto the stack, in the second half it may pop off the from the stack, until it's empty). There is *no way* that the parser can know, by looking at the next intput(s), when the time to switch from pushing to popping has come (as there are *only* **b***'s* on the input unlike in the other 2 grammars where a **c** demarcated when it's time to parse "the second half" of **b**'s).

(c) The question about *regularity* is about the respective *languages*, not the grammars. The lecture did not cover technical background to formally *prove* that a given language is *not* regular.[4] What is covered in the lecture is the general fact that regular languages are a strict subset of context-free language (and those in turn a a strict subset of the context-sensitive ones . . . ). This general relationship between language is the *Chomsky-hierarchy.*

The question here is to informally know where the border between regular languages and context-free languages lies, and to argue (if that's the case) why a given language is *not regular.* If the given language is *regular,* the straighforward and expected answer is to give the regular expression which represents the language.

(i) The first grammar $G_1$ produces the language where there is a number of **b**'s followed by *the equal number* of **c**. In more formulaic notation:

$$\mathcal{L}(G_1) = \{\mathbf{b}^n\mathbf{c}^n \mid n \geq 0\}$$

As a general intuitive explanation of what regular languages (or finite-state automata) cannot do is: they can produce "*many*" symbols, but they cannot (arbitrarily) *count how many* and *remember* the number. Of course, the language containing

"15 **a**s followed by 15 **b**s"

is fine, because the number is *fixed* (one just needs enough states, approximately 15+15=30 states). The language of $G_1$ is different, first $\mathbf{b}^n$ needs to be produced (or scanned), and, after seeing the first **c**, the automaton must have "*remembered*" the number $n$. Since the number depends on the word and can be arbitrarily large, this cannot be done by a finite-state automaton.[5] Note also: it's *characteristic* for CF languages that they capture such "nested balancing" structures ("each **b** to the left is matched by one **c**"). Many examples on the lecture dealt with various nestings of parentheses and other syntactic structures. Indeed, nested parenthetic structures or nesting of "structures" in general is almost synonymous with the *syntactic structure* of programming languages: blocks can be nested and each "`begin-block`" must have a matching "`end-block`", each opening "**(**" must have one maching "**)**" . . . . Of course, the language of "simple parenthesis" from the lecture exactly corresponds to the language here (just with different symbols).

There "nesting" or "parenthetic" structures in strings of terminal symbols are the aspects which are done by the parser, and which *cannot* be done by the lexer.

Finally: it should be noted that an argument based *only* on the *form* of the grammar is not good. It has been mentioned that *left-linear* grammars generate regular languages (analogously for right-linear grammars, but not for grammars which contains a mixture of left-linear and right-linear productions). Now, grammar $G_1$ is neither left-linear nor right-linear. But, as mentioned, the question here is about the generated *language* not the form of the grammar.[6]

(ii) This grammar is neither left-linear nor right-linear. However, that does not answer the question, we have to look at the language (as discussed). The language $\mathcal{L}(G_2)$

[4] The most relevant result in that context is known as the *pumping lemma* (for regular languages). As said, this lemma is not part of the pensum.

[5] We have not formally introduced *push-down automata,* which are automata with a stack, they were shortly mentioned on connection with the Chomsky-hierarchy. Pushdown automata can use the additional unbounded stack memory exactly for that. Note also: the parsing process for the various classes done in the lecture —LR(0), SLR . . . — *did* use a *stack* explicitly in the case of bottom-up parsing. For top-down parsing, the recursive procedures underlying the recursive descent parsing approach of course implicitly contain a stack was well.

[6] That is kind of question different from questions like "is the grammar left-linear" or "is the *grammar* SLR". Of course one may also ask "is the following *language* SLR" . . . , but that's a different from and harder than asking analgously about a grammar. As a final remark: it also means: if one has determined that the language of a grammar happens to be regular, that fact cannot be use to short-cut the question whether the grammar is LR(0) etc.

consists of an *odd-numbered* amount of **b**'s, in short:

$$\mathcal{L}(G_2) = \{\mathbf{b}^{2n+1} \mid n \geq 0\} \ .$$

That's rather easy to capture by a regular expression (or a FSA), for instance as follows:

$$\mathbf{b}(\mathbf{bb})^* \ .$$

(iii) The language here can be characterized as follows

$$\mathcal{L}(G_3) = \{\mathbf{b}^n\mathbf{c}\mathbf{b}^n \mid n \geq 0\}$$

and is not regular for the same reasons as the language of $G_1$.

(d) The parsing table is given as follows.

| state | input | | | goto |
|---|---|---|---|---|
| | **b** | **c** | **$** | $A$ |
| 0 | $s:2$ | | $r:(A \to \epsilon)$ | 1 |
| 1 | | | accept | |
| 2 | $s:2$ | $r:(A \to \epsilon)$ | $r:(A \to \epsilon)$ | 3 |
| 3 | | $s:4$ | | |
| 4 | | $r:(A \to \mathbf{b}A\mathbf{c})$ | $r:(A \to \mathbf{b}A\mathbf{c})$ | |

Table 2: SLR(1) parsing table for $G_1$

| *stage* | parsing stack | input | action |
|---|---|---|---|
| 1 | $\$_0$ | **bbcc $** | shift: 2 |
| 2 | $\$_0\mathbf{b}_2$ | **bcc $** | shift: 2 |
| 3 | $\$_0\mathbf{b}_2\mathbf{b}_2$ | **cc $** | reduce: $A \to \epsilon$ |
| 4 | $\$_0\mathbf{b}_2\mathbf{b}_2A_3$ | **cc $** | shift: 4 |
| 5 | $\$_0\mathbf{b}_2\mathbf{b}_2A_3\mathbf{c}_4$ | **c $** | reduce: $A \to \mathbf{b}A\mathbf{c}$ |
| 6 | $\$_0\mathbf{b}_2A_3$ | **c $** | shift: 4 |
| 7 | $\$_0\mathbf{b}_2A_3\mathbf{c}_4$ | **$** | reduce: $A \to \mathbf{b}A\mathbf{c}$ |
| 8 | $\$_0A_1$ | **$** | accept |

Table 3: Parser run (reduction) for $G_1$ and input **bbcc**

□

**Exercise 11 (Classes and virtual tables (20%))**

Assume we are dealing with an OO language where a *virtual* method in a class can be redefined ("overriding") in subclasses of that class. A virtual method is declared via the `virtual` modified, where a *re*definition is declared with the modifier `redef`. Methods without `virtual` modifier are "ordinary" methods and cannot be redefined. Note that it's not completely as in Java. In Java, all methods are virtual, whereas here, that's only the case for methods with `virtual` modifier.[7] Consider the following classed, defined in that assumed language

[7] At that point it's unclear if `redef`-methods may be redefined *again*.

```
class A {
  virtual void m (int x, y) { ... }
  void p () { ... }
  virtual void q() { ... }
}

class B extends A {
  redef void m (int x, y) { ... }
  void r() { ... }
}

class C extends A {
  redef void q() { ... }
}

class D extends B {
  redef void m (int x, y) { ... }
}

class E extend B {
  redef void q() { ... }
}

class F extends C {
  redef void m(int x, y) { ... }
}
```

(a) We assume first that the class for a given object determines, in the standard way, which version of a virtual method is being called.

Do the *virtual tables* for the all the classes `A`, `B`, ..., `F`. For each element in the table, use the notation `A::m` to indicate which method actually is meant. The indices in this tables are supposed to start with 0.

(b) For the rest of this problem, we assume the following semantics: A refined virtual methods, say `m`, *first* executes the correspoidng virtual or redefined method (i.e., $m$) in the closest superclass containing such a method, *before* executing its own body. This in turn may leads to the situation that redefined or virtual methods `m` in *further* superclasses are executed.

One can implement that by setting in the right call as *first* statement in the body of *redefined* methods. However: the semantics of *parameter passing* here is assumed to be a little by specific in that the straightforward way won't work. The parameters handed over in the original call should go directly as parameters to the method which is being executed first, i.e., the one which are marked `virtual` in the program. When that is finished executing, the values which are contained in that versions parameters be transferred a actual parameters for the next deeply nested redefined method, etc. As a consequence, the stack of the call must be set-up first, and that the actual parameterrs must handed over to the first virtual method which is supposed to be executed.

As example: asume `m` is called with `m(1,2)` on a `D`-object. In that case the stack is being set up and the actual parameter go into the activation recode corresponding to `A::m`, and the execution can start executing `A::m`. Upon exit of `A::m`: the values of `x` and `y` will be handed over as actual parameters to the version of `m` which is supposed to be executed next.

In order to implement this new semantics, we need to extend the virtual tables in such a way that for each index, a *list* of methods is available. This list will, consequently, give the *sequence* of methods which will be called.

*Draw* these new virtual tables for classes `D` and `F`. The tables for `B` and `C` are given in Table 4. To indicate methods, use the same notation as before.

□

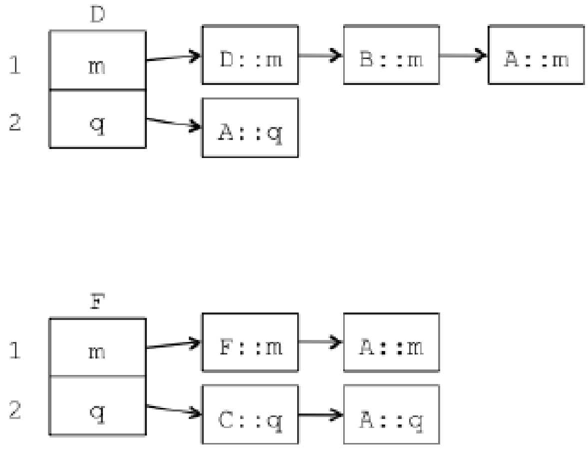



Figure 4: Extended virtual tables for `B` and `C`

**Solution:**

(a) The virtual tables are shown in Table 5.

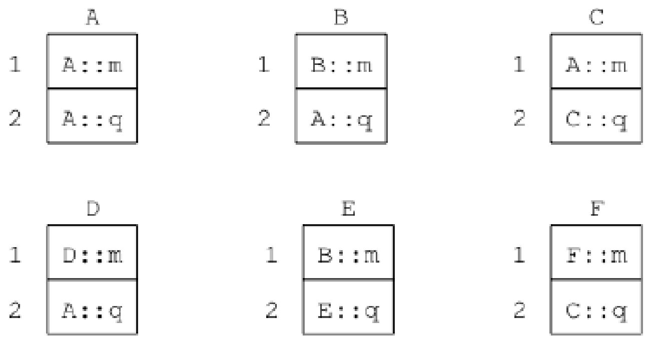



Figure 5: Virtual tables for the given class hierarchy

(b)

□

**Exercise 12 (Attribute grammars (30%))**

The following is a fragment of a grammar for a language with classes. A class *cannot* have superclass; instead it must implement one or more interfaces.

$$
\begin{array}{rcl}
\textit{class} & \rightarrow & \textbf{class name implements } \textit{interfaces}\,\{\,\textit{decls}\,\} \\
\textit{decls} & \rightarrow & \textit{decls}\ \textbf{;}\ \textit{decl} \mid \textit{decl} \\
\textit{decl} & \rightarrow & \textit{variable-decl} \mid \textit{method-decl} \\
\textit{method-decl} & \rightarrow & \textit{type}\ \textbf{name}\ \textbf{(}\,\textit{params}\,\textbf{)}\ \textit{body} \\
\textit{type} & \rightarrow & \textbf{int} \mid \textbf{bool} \mid \textbf{void} \\
\textit{interfaces} & \rightarrow & \textit{interfaces}\ \textbf{,}\ \textit{interface} \mid \textit{interface} \\
\textit{interface} & \rightarrow & \textbf{name}
\end{array}
$$

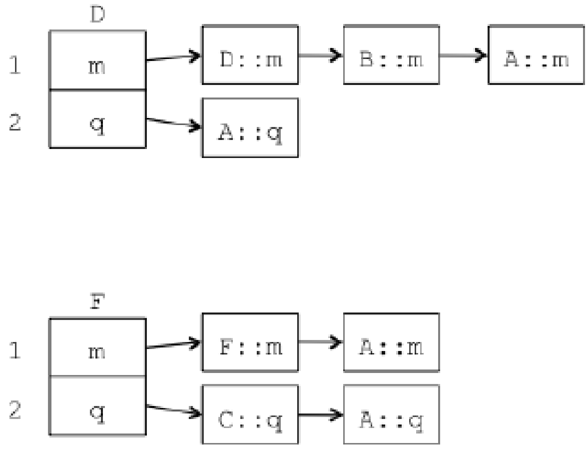



Figure 6: Extended virtual tables for `D` and `F`

The words in *italics* are non-terminals, those in **bold-face** are terminals, and **name** represent names handed over by the scanner. That terminal **name** has an attribute "`name`" (a string).

A special feature of this language is that class methods with the same name as the *interfaces* the class implements are *constructors* for the class. A class can thus contain more than one method with the same name as one of the implemented interfaces, also with different parameter. The latter, though, is not the topic of the problem here.

The generation of new objects is of the form

$$\textbf{new}\langle \textit{classname} \rangle \,\mathbf{.}\langle \textit{interface} - \textit{name} \rangle (\langle \textit{actual} - \textit{parameters} \rangle)$$

since different classes can implement the same interface.

One *requirement* of this language is that constructor need to be specified with the type **void**, and that's the requirement which you are requested to check using *semantical rules.* Thus: give semantical rules in the following fragment of an attribute grammar. In the definition, you can use functions and set etc you need, but you need to define them properly.

Answer with question using the corresponding attachement.

| Grammar Rule | Semantic Rule |
|---|---|
| *class* → **class** ***name*** ***implements*** *interfaces* **{** *decls* **}** | |
| $decls_1$ → $decls_2$ ; *decl* | |
| *decls* → *decl* | |
| *decl* → *method-decl* | |
| *method-decl* → *type* ***name*** **(** *params* **)** *body* | |
| *type* → **int** | *type*.type = int |
| *type* → **bool** | *type*.type = bool |
| *type* → **void** | *type*.type = void |
| $interfaces_1$ → $interfaces_2$, *interface* | |
| *interfaces* → *interface* | |
| *interface* → ***name*** | *interface*.interfaceName = ***name*** |

□

**Solution:**

| Grammar Rule | Semantic Rule |
|---|---|
| *class* → **class name implements** *interfaces* **{** *decls* **}** | *decls*.setOfInterfaceNames = *interfaces*.setOfInterfaceNames |
| $decls_1$ → $decls_2$ ; *decl* | $decls_2$.setOfInterfaceNames = $decls_1$.setOfInterfaceNames<br><br>*decl*.*setOfInterfaceNames* = $decls_1$.setOfInterfaceNames |
| *decls* → *decl* | *decl*.*setOfInterfaceNames* = *decls*.setOfInterfaceNames |
| *decl* → *method-decl* | *method-decl*.*setOfInterfaceNames* = *decl*.setOfInterfaceNames |
| *method-decl* → *type* **name** **(** *params* **)** *body* | **if** *method-decl*.setOfInterfaceName.*has*(**name**.name) **then if** (**not**(*type*.type = void)) **then** *error*("constructor not of type void") |
| *type* → **int** | *type*.type = int |
| *type* → **bool** | *type*.type = bool |
| *type* → **void** | *type*.type = void |
| $interfaces_1$ → $interfaces_2$, *interface* | $interfaces_1$.setOfInterfaceNames= $interfaces_2$.setOfInterfaceNames + [*interface*.interfaceName] |
| *interfaces* → *interface* | *interfaces*.setOfinterfaceNames.insert(*interface*.interfaceName) |
| *interface* → **name** | *interface*.interfaceName= **name** |

□

### Exercise 13 (Code generation & P-code (25%))

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

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

  (ii) Sketch which *data structures*

(b)

| | | |
|---|---|---|
| `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 4: P-code instructions

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

Consider the P-instruction

```
ldv b
```

where $b$ is a variable whose value resides in the home position. This instruction therefore pushes the value of $b$ onto the top of the stack. When translating that to machine code, a question there is what is better: 1) doing a `LOAD` instruction so that the value of $b$ ends up in register or alternatively 2) push a descriptor onto the stack marking that $b$ resides in its home position.

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

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

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

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

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

**Solution:**

(a) **!!!!**

(i)

(ii)

(iii)

(b)

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

(ii) the situation leaves room for many optimizations. One situation discusses is that if the expression contains a function call (or method call etc). I would not subsume that in this tasks, since 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. In therefore the expression is *side-effect free* there's no need to load the value directly, as it effectively does not matter when it's loaded. Therefore one may be better off simply using the descriptor stack marking where the variable is being found in memory.

(d) In any case we need the following

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

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

It should be noted that the descriptors stack is not really good enough to keep track of all the information the code generator wants to keep an eye on. At least if it wants to keep a level of overview over registers and variables comparable to the code generator from the lecture. The reason why the stack itself is not good for that, no matter how much info we plan to store into the stack entries, is simply that poping 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* 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.

# 7 2012

**Exercise 14 (Context-free grammars and parsing (25%))**

Consider the following grammar $G_1$:

$$S \rightarrow \mathbf{a} \mid S \# S \mid S @ S$$

Here, $S$ is the start symbol and the only non-terminal. The symbols **a**, #, and @ (and the end-of-input symbol **$**) are terminals.

(a) Give a concrete argument why the grammar is ambiguous.

(b) Assume that

- the operator # has *low* precedence and is *right*-assosicative
- the operaotr @ has *high* precedence and is *left*-associative

Give a new grammar $G_2$ which describes the same language as $G_1$ and follows the rules just given. You may introduce new non-terminals, and it's not required to give arguments that $G_2$ is unambiguous beyond pointing out similarities of corresponding unambiguous grammars from the pensum.

(c) We look at the grammars $G_1$ and $G_2$, as well as the following grammar $G_3$ (where the latter contains + as new terminal symbol)

$$S \rightarrow \mathbf{a} \mid S \# S \mid S @ S \mid + S +$$

Which of the languages $\mathcal{L}(G_1)$, $\mathcal{L}(G_2)$, and $\mathcal{L}(G_3)$, are *regular* and which not. Explain and give a regular expression for those which languages which happen to be regular.

(d) Give the LR(0)-DFAs for the ambiguous grammar $G_1$ (using a $S'$ in the usual way).

(e) Give the *First* and *Follow*-sets of $S$ in $G_1$ (making the usual use of the symbol **$**). Indicate which states from the DFA of the previous sub-problem have

(i) conflicts which *cannot* be resolved with LR(0)-criteria, but *can* be solved via SLR(1)-criteria. Explain.

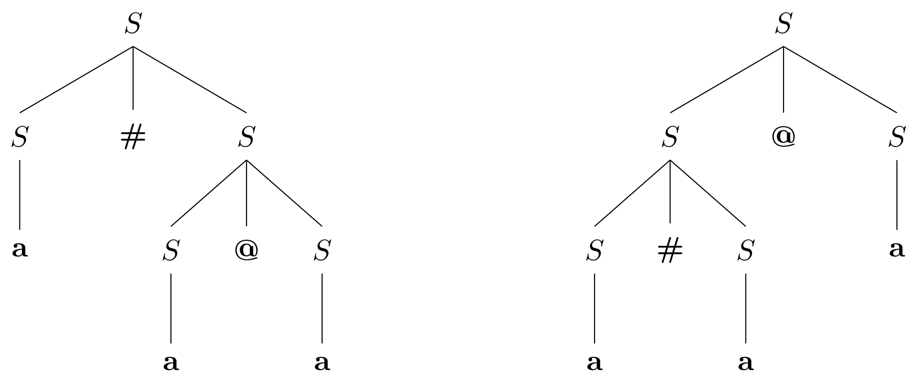
(ii) Conflicts which *cannot* be resolved by SLR(1)-criteria. Explain.

(f) For the "conflict"- states of the automaton from point (ii) of the previous sub-problem: explain how you would solve them "manually" in order to obtain the precendeces and associativities as given in sub-problem (b)

(g) Give the SLR(1)-parsing table for $\mathcal{L}(G_1)$, using the answers from subproblems (d) and (f). The table thus should have have maximally *one* action per slot and the resulting syntax analysis should follow the rule from sub-problem (b). □

**Solution:**

(a) "Concrete" means: give one sentence that has two different derivation trees (and give them). Well, we have two binary operations (and the grammar does not specify any *precedence* (like that @ has a higher priority/precedence/binding power than #). That's the root of ambiguity. Two different trees for the same word are:




(b) Here's two possible solutions:

$$
\begin{array}{rcl}
S & \rightarrow & T \# S \mid T \\
T & \rightarrow & T @ F \mid F \\
F & \rightarrow & \mathbf{a}
\end{array}
\qquad\qquad
\begin{array}{rcl}
S & \rightarrow & T \# S \mid T \\
T & \rightarrow & T @ \mathbf{a} \mid \mathbf{a}
\end{array}
$$

They are built according to the principles as in the lecture, see also [Louden, 1997, Section 3.4.2].

(c) $G_1$ and $G_2$ are the same language (provided the first subproblem was solved correctly ...). They are regular and a corresponding regular expression is:

$$\mathbf{a}((\# \mid @)\mathbf{a})^*$$

For $G_3$, the +-signs are treated in the form that the number of + "generated" left of the $S$ equals the number of + right of $S$. That's prototypical for non-regular languages. See also the corresponding explanations in the exam from 2011. Of course the language here does not just contains +'s but also #'s or @'s but the basic fact that a finite-state automaton cannot count symbols unboundedly and remember the number applies also here (for the +'s), thus a FSA cannot recognize $\mathcal{L}(G_3)$ and the language is therefore not regular.

(d) The LR(0)-DFA is given in Figure 7

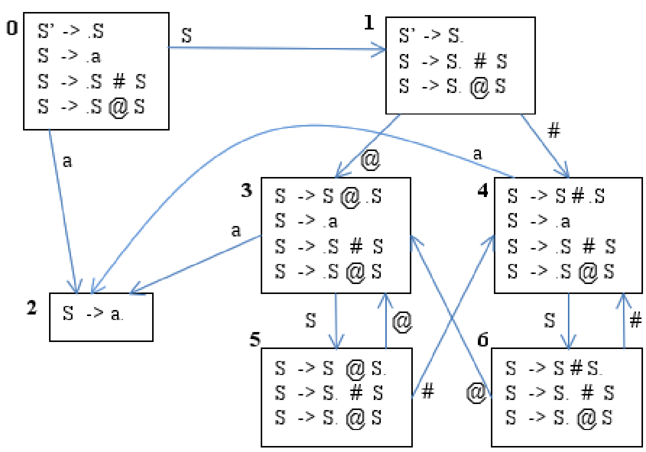



Figure 7: LR(0)-DFA

(e) The correspondings sets for $S$ are as follows:

$$First(S) = \{\mathbf{a}\} \quad \text{and} \quad Follow(S) = \{\#, @, \$\}]$$

(i) There is one LR(0) conflict in state 1, which can be solved with the SLR(1)-criterion (taking the follow-sets into account). The state *accepts* for **$**, and shifts for # and **@**.

(ii) There are additional LR(0) conflicts in both 5 and 6

As a side remark: as the grammar is not ambiguous, there *have* to be conflicts which are not solvable via SRL(1) (and of course also not via LR(0))

(f) We have to look at the 2 states which have SLR(1) conflicts

(i) state 5: In this state, the string

$$S\,@\,S$$

is on top of the stack.[8] The options now are #, **@** or **$**; only the first two situation constitute a conflicts, as found out in the previous sub-problem, and those we have to disambiguate.

In case of an #: In this case, we *reduce*. As mentioned, **@** binds stronger than # (has higher precendence), and a **@**-binary expression it is currently on top of the stack, choosing a reduce step in this situation realizes that higher precedence. Remember that a reduce-step conceptually builds up one new node in the parse-tree which is growing in a *bottom-up* manner.

In case of a **@**: Now it's a question not of precedence but of *associativity*. The **@**-symbol is specified to be *left*-associative. Therefore, we need to *reduce* also in this case.

The last case of **$** also results in a reduce, but that was already clear.

(ii) For state 6: in this case,

$$S\,\#\,S$$

is on top of the stack. With analogous argumentation than for state 5, we have to *shift* for # (in contrast to **@**, the # is *right*-associative)! For **@**, we need to shift, because this time, the next symbol (i.e., **@**) has a higher precedence compared to that of the symbol on the stack (i.e. #).

(g) The table looks as follows:

| | a | # | @ | $ | S |
|---|---|---|---|---|---|
| 0 | s2 | | | | 1 |
| 1 | | s4 | s3 | acc. | |
| 2 | | r(S -> a) | r(S -> a) | r(S -> a) | |
| 3 | s2 | | | | 5 |
| 4 | s2 | | | | 6 |
| 5 | | r(S->S @ S) | r(S->S @ S) | r(S->S@S) | |
| 6 | | s4 | s3 | r(S->S # S) | |

□

**Exercise 15 (Run-time environments (25%))**

In this task we are given a Java-like language where methods can have locally defined methods. Furthermore it is possible to declare variables and methods at the othermost program level. That is supposed workd as usual in languages with static scoping.

The following is a program in this language.

[8] That can be seen by following the paths from the initial state 0 to that state 5. Note that there is not only one path, but many (actually infinitely many). All of them however, leave the indicated word on top of the stack.

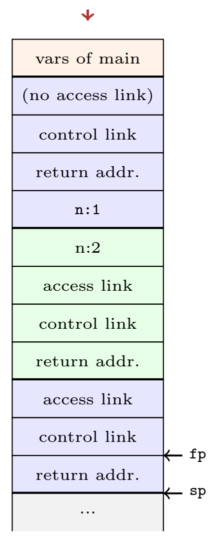
```
{
    class C {
        void m1 () {
            void f() {};

            f ();
        }
        void m2 () {
            int i;
            void g() {
                int j;
                j = i;
            };
            i = 1;
            rC.m1();
        };
    };

    C rc ;
    void main () {
        rc = new C{} ; rC.m2{};
    }
}
```

Draw the call stack in the situation where the activation record for `f` is on top of the stack for the first time. Draw the stack including variables, access-lonks, and control-links, but without access-links for methods which are directly declared in a class (one can assume for this that the access-link point to the `C`-object, but this is not important for the task at hand). □

**Solution:**




(a) calls m → p → r → q

□

# 8 2013

**Exercise 16 (CFG and parsing (35%))**

Consider the following 2 grammars $G_1$ and $G_2$:

$$S \rightarrow \textbf{(}\, S \,\textbf{)} \mid \boldsymbol{\epsilon}$$

$$S \rightarrow \textbf{(}\, S \,\textbf{)} \mid \mathbf{a}$$

$S$ is the only *non-terminal* and thus also the start symbol. The symbols **(**, **)**, and **a** are *terminals* (together with **$**, which has the usual meaning).

(a) Which of the languages $\mathcal{L}(G_1)$ and $\mathcal{L}(G_2)$ are *regular*? For those which are regular, give a regular expression representing the language.

(b) Next we look at a slighty more complex grammar $G_3$:

$$\begin{array}{lcl} A & \rightarrow & \textbf{(}\, S \,\textbf{)} \mid \textbf{(}\, B \,\textbf{]} \\ B & \rightarrow & \;S \;\mid \textbf{(}\, B \\ S & \rightarrow & \textbf{(}\, S \,\textbf{)} \mid \boldsymbol{\epsilon} \end{array}$$

Now, $A$, $B$, and $S$ are *non-terminals* with $A$ as start symbol. The symbols **(**, **)**, and **]**, are *terminals* (together with **$**, which has the usual meaning).

Give *4 sentences* of the language $\mathcal{L}(G_3)$ such that they, in the best possible manner, cover the different "kinds" of sentences from the language $\mathcal{L}(G_3)$. Describe additionally in words the sentences from $\mathcal{L}(G_3)$

(c) For $G_3$, determine the first and follow-sets for $A$, $B$, and $S$. Make use of $\boldsymbol{\epsilon}$ as in the book. Just give the result, no need for explanation.

(d) Draw the LR(0)-DFA for grammar $G_3$, after having introduced a new start symbol $A'$, as usual. Hint: there are approximately 10 states, and 2 of them contain 6 items. Be precise not to forget any elements in the closures when building the state, and combine equal states.

(e) Put numbers on the states, starting from 0. Consider all states and discuss shortly those states which have (at least) one LR(0)-conflict. Which one of those have *also* and SLR(1)-conflict. Is $G_3$ an SLR(1)-grammar?

(f) Draw *parts* of the parsing table for $G_3$ according to the SLR(1)-format, namely those 2 lines which correspond to the states of the automaton which contain 6 items. If $G_3$ is not SLR(1), give all alternatives in the slots where there is an SLR(1)-conflict. Take care not to forget any of the "symbols" needed in the header-line of the table.

**Solution:**

(a) Both langagues are *not regular*. For both, the reason is that the language captures "well-balanced" parenthethic structures or nesting (here, actual parentheses). An FSA cannot parse (or generate) such structures, as it only has finite memory. For more and deeper explanations, see exercise 10 from 2011.

(b) Possible sentences (= sequences of terminals) are, for instance

**( )**
**( ( ( ( ( ( ) ) ) ) ) )**
**( ( ( ( ( ) ]**
**( ]**

It's hard to pinpoint what exactly is the best possble selection of sentences, but one should avoid having two examples of the "same pattern" (like having **( ( ) )** *and* **( ( ( ) ) )** as illustration). Furthermore, "extremal cases" may capture the spirit of the grammar (like having **( )** and **( ]**) and making sure that one covers all (or enough different) productions.

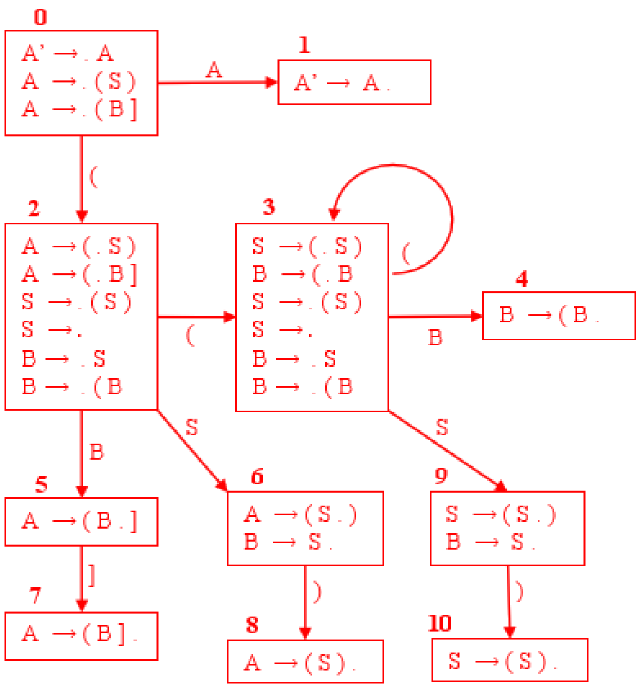
A possible rendering in words could be

> The sentences start with *one or more* **(**'s. Say the number is $n \geq 1$. The sentence then is finished by the *same* number $n$ of **)**'s, or else finished by a number of **)**'s which is strickly smaller than $n$ (possibly 0) followed by one **]**.

(c) The standard use of $\epsilon$ is to indicate in the *First*-set, that the corresponding symbol is *nullable*.[9]

| non-term. | *First* | *Follow* |
|---|---|---|
| $A$ | **(** | **\$** |
| $B$ | $\epsilon$, **(** | **]** |
| $S$ | $\epsilon$, **(** | **)**, **]** |

(d) The LR(0)-DFA looks as follows:




[9]Technically, in order to do the "closure" *algorithm* of iteratively calculating the first-set of a non-terminal, it's necessary, that the output to a call to *First* no only gives the set of first terminal symbols, but also information whether or not the symbol is nullable (note that $\epsilon$ is *not* a terminal symbol ...). Indicating nullability is done conventionally by adding $\epsilon$ to the result of *First*. Even if this task here does *not* require to actually do the *First*-algo step-by-step, still a correct answer must indicate nullability with $\epsilon$.

(e) States with LR(0)-conflicts are the following four states: 2, 3, 6, 9. Those are among the states containing a *complete item* (actually, the same complete item in all 4 cases). That indicates that a *reduce* step is possible, but those states also allow shift-step(s). This means: those states have LR(0)-conflicts.

To check if they also suffer from SLR(1)-conflicts, we need to consult the follow sets, to be precise, the follow-sets of $S$ (which consists of **)** and ]) for states 2 and 3, and the follow-set for $B$ for the other 2 states.

- for states 2 and 3: shift is done there with **(**, but reduction using production $S \to \epsilon$ is done only for ] and **)**, so there's no confusion possible here. Therefore the states have no SLR(1)-conflict
- for states 6 and 9: A shift is doable (only) for **)**, but one can only reduce for ], thus also those states are ok, SLR(1)-wise.

(f) The parsing table for the states 2 and 3 looks as follows:

| | ( | ) | ] | $ | A | B | S |
|---|---|---|---|---|---|---|---|
| 2 | s3 | r(S → ε) | r(S → ε) | | | 5 | 6 |
| 3 | s3 | r(S → ε) | r(S → ε) | | | 4 | 9 |

□

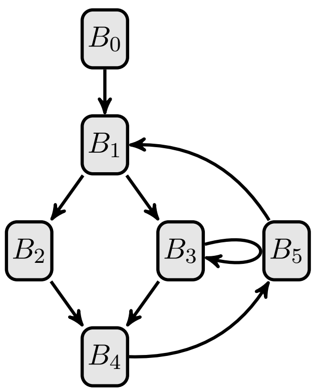
**Exercise 17 (Code generation and analysis (25%))**

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

**Solution:**

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

i. the variable is not *used* later in the block.
ii. $x$ is not contained in $outLive(B)$.

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

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

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

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

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

Analogously for the second set $\{B_1, B_2\}$.

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

(c) Trivial things first: to move it out of the loop means to move it *before* the loop (not afterwards), obviously. The canonical place thus is *immediately* before the loop we are moving out of. As we are dealing with *loops* in the specific sense discussed (as opposed to general cycles in a graph), there is exactly one well-defined entry point to the loop, and that is exactly where the code needs to be moved to. More precisely, it need to be moved immediately *before* that node. In our example, the entry node of the "big" loop is $B_1$ and the predecessor outside of the loop is $B_0$. To place the code, one simply introduces a new block, say $B_6$, placed *between* $B_0$ and the loop's entry node $B_1$. In particular, the code cannot be placed inside $B_1$ (at the beginning, say)[10] and the arc back from $B_5$ still has $B_1$ as successor, and not the new node.

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

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

```
//---------------------- e = a - b
MOV a R0
SUB b R0   // e ∈ r0, ''all'' reg's full
// --------------------- f = a - c
MOV R0 e   // f has a next-use, so, clear
           // the only register r0
MOV a R0   //
SUB c R0   // f ∈ r0
// --------------------- d = f - e
MOV R0 f
SUB e  R0   // f is live after the block
            // and must therefore be saved
            // f before the SUB step is already
            // in the right place (in r0)
            // afterwards, d is in r0
//---------------------- end-of-basic block
MOV R0 d    // save value for d back to main memory
            // all other variables are already up-to
            // data in their resp. ''home positions''
```

# 9 2016

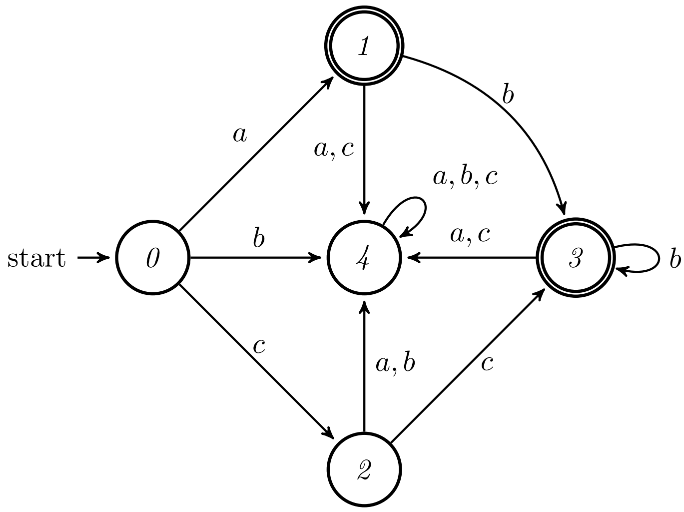
**Exercise 18 (Regular expressions and scanning (1%))**

(i) Let $\Sigma$ 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 $\Sigma$ *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?

(ii) 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.




(iii) 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:

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 $\Sigma$ 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).

(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.

$$/ o\ (o^* z \mid /)^* o^+ / \tag{4}$$
$$/ o\ /^* (o^* z\ /^*)^* o^+ / \tag{5}$$
$$/ o\ (o^* z\ /^*)^* /^* o^+ / \tag{6}$$
$$\tag{7}$$

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.

□

**Exercise 19 (Context-free languages and parsing (%))**

(i) Consider the following context-free grammar:

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

In the grammar, **x** and **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.

(ii) 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.

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

grammar $G_1$:
$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow A\mathbf{a} \mid \epsilon \end{aligned}$$

grammar $G_2$:
$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow \mathbf{a}A \mid \epsilon \end{aligned}$$

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 5 and the DFA is given in Figure 8.

Table 5: First- and follow-sets

| non-term. | *First* | *Follow* |
|---|---|---|
| $S'$ | **x** | **$** |
| $S$ | **x** | **$**, **y** |
| $A$ | **x** | **y** |
| $B$ | **y** | **$**, **y** |

(ii) The grammar is *not* SLR(1). That can be seen in state 7: Since *Follow* $B$ contains **y** (a terminal which follows the "parser position **.**" in one item), there's a shift-reduce conflict on symbol **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).

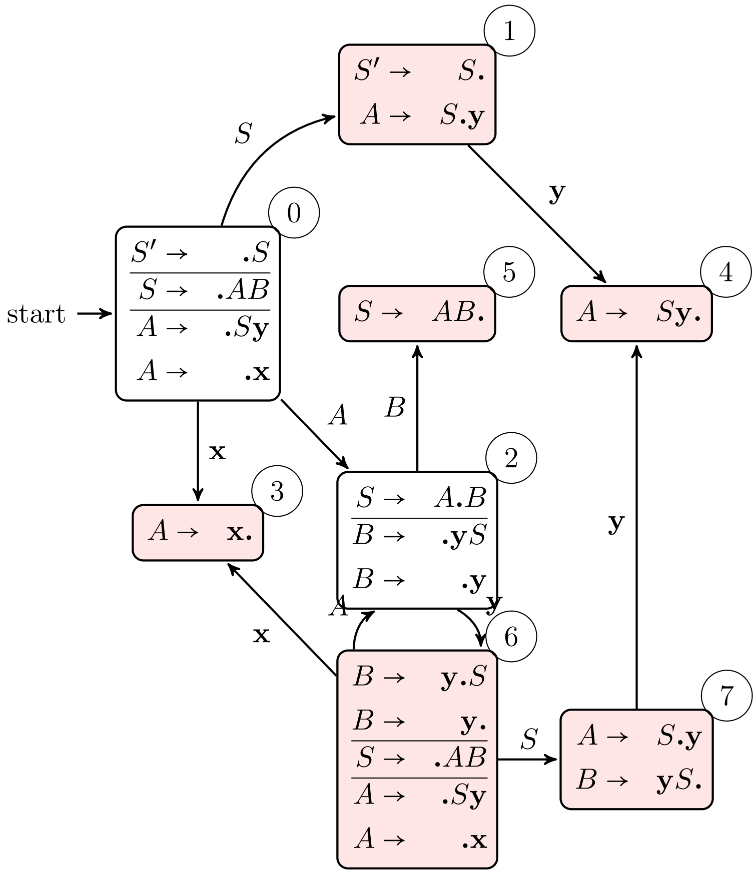



Figure 8: LR(0)-DFA

(i) *Problematic* for LR(0) are $\epsilon$-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:

$$A \to \epsilon \mid \mathbf{a} \tag{8}$$

For LL(1) parsing, $\epsilon$-productions are unproblematic.[11] 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:

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

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

Of course the *left-recursive* alternative

$$A \to \epsilon \mid A\mathbf{a} \tag{10}$$

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

[11] 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 $\epsilon$ productions.

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

$$A \to ab \mid ac \tag{11}$$

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 9.

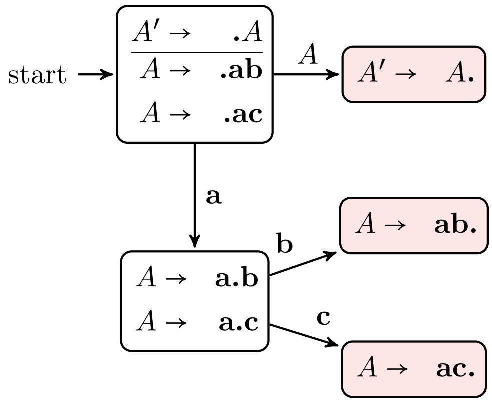



Figure 9: LR(0)-DFA

Remark: As mentioned, left-recursion is problematic for LL(1), as well. Thus, one might be tempted to use (10) as an example. It's certainly not LL(1) but unfortunately, the grammar is also *not* LR(1) (still containing an $\boldsymbol{\epsilon}$-production).

Additional remark: Even if we replaced the $\boldsymbol{\epsilon}$ with a terminal **b** yielding

$$A \to \mathbf{b} \mid A\mathbf{a} \tag{12}$$

the grammar won't be LR(1):

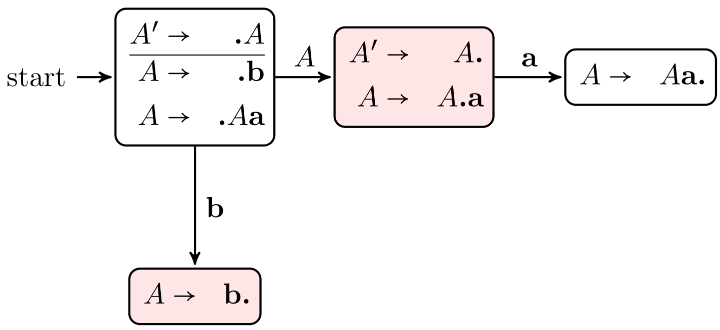



Figure 10: LR(0)-DFA ("**ba***")

(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, 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 $A \to \mathbf{a}A$) 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:

```
         A              A
        / \            / \
       A   a          a   A
      / \                / \
     A   a              a   A
    /                        \
   ..                         ...
   A                           A
  / \                         / \
 e   a                       a   e
```

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).

□

**Exercise 20 (Attribute grammars (%))**

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.

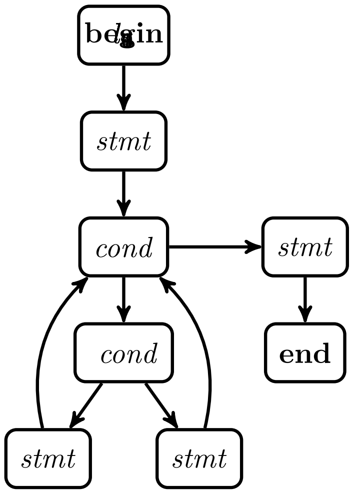
**Starting point: attribute `label` given** We shall assume that the non-terminals *assign* and *cond* as well as the terminals **begin** and **end** all carry an an attribute `label`, containing a *label* value for indentification. These label values are *already filled in.* So you can make use of, 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 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`}<br>**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* 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

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

Table 6: Overview over attributes

that. What is not depicted in the picture are *assign*-non-terminals, [12] 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* → *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 6. 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 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.[13]

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

[12] 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.

[13] 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.

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

Figure 11: AGrammar for `first` and `lasts`

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 11. 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 12. Now the perspective changes: 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.

**Exercise 21 (Code generation (%))**

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

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

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

Figure 12: AGrammar for `succ`

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

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

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

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

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

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

**Solution:**

(a) Register descriptors:

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

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

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

For a correct answer, one should give

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

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

**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 6 and 7 makes use of that.

Listing 6: Reuse of a register for $y$

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

Listing 7: Reuse of register no longer possible

```
   // initially, R empty
-------------------------------------------------------------
   y := x + 1   // use R for the result:
                // Load x:                                  1
                // R |-> y (not up-to date)
   a := t1 + t2 // Store R -> y   (get-reg-(3)              1
                // Load t1                                  1
                // Load t2                                  1
                // R |-> a  (not up-to date)
   z := y + 1   // Store a (no reuse)                       1
                // Load y                                   1
                // result: R <- z (not up-to date)

                // end of block: store z                    1
-------------------------------------------------------------
```

□

# References

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

[Louden, 1997] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.