Universitetet i Oslo
Institutt for Informatikk

Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2018

Collection of older exam questions

14. 05. 2018

Issued: 14. 05. 2018

Contents

1 2005 **2**
Regular expressions and automata . . . 2
context-free grammars and parsing . . . 2
Attribute grammars and type checking . . . 3

2 2006 **6**
Parameter passing and attribute grammars . . . 6
Context-free grammars and parsing . . . 7
Classes and virtual tables . . . 8

3 2007 **10**
Code generation . . . 10

4 2009 **10**
Code generation . . . 10

5 2010 **11**
Code generation . . . 11

6 2011 **12**
CFGs and Parsing . . . 12
Classes and virtual tables . . . 12
Attribute grammars . . . 13
Code generation & P-code . . . 15

7 2012 **16**
Context-free grammars and parsing . . . 16
Run-time environments . . . 17

8 2013 **19**
CFG and parsing 19
Code generation and analysis 19

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.

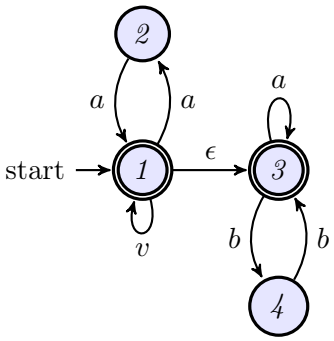
1 2005

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.*

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

[1] Especially in the footnotes.

Consider the following grammar G_1:

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

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:

$$F \rightarrow \texttt{+}\ F \mid \texttt{-}\ F \mid \textbf{num}$$

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

- - 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. □

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

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

class	→	**class name** *superclass* **{** *decls* **}**
decls	→	*decls* **;** *decl* | *decl*
decl	→	*variable-decl*
decl	→	*method-decl*
method-decl	→	*type* **name (** *params* **)** *body*
type	→	**int** | **bool** | **void**
superclass	→	**name**

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:

```
1  class A {
2    int i;
3    void P { ... AP ... };
4    void Q { ... AQ ... };
5  }
6
7  class B extends A {
8    int j;
9    void Q { ... BQ ...};
10   void R { ... BR ...};
11 }
12
13 class C1 extends B {
14   void P { ... C1P ...} ;
15   void S { ... C1S ...} ;
16 }
17
18 class C2 extends B {
19   int k
20   void R { ... C2R ...} ;
21   void T { ... C2T ...} ;
22 }
```

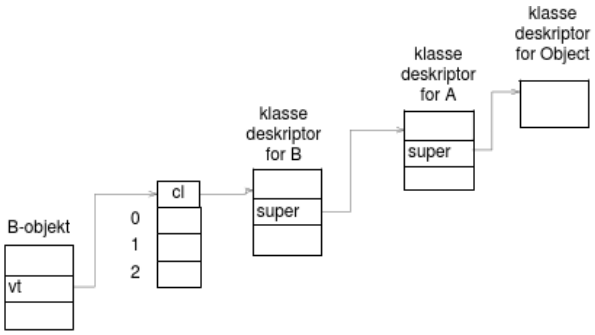
Show how objects of classes C_1 and C_2 are structure (show their layout) and draw the virtual function table[2] for each of the classes. Use the "names" shown in the above method bodies to indicate elements in the virtual function tables.

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

refExpr **instanceof** *class*

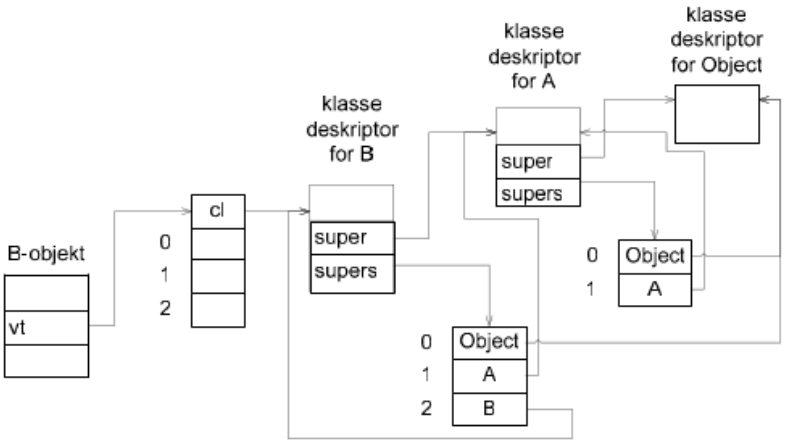
is "true" if the object pointed at by *refExpr* is of a class which is not "null", and which is class *class* or a subclass of *class*. Otherwise, the value of the expression is "false".

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:

[2] name

```
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)
```

□

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	→	**proc id** (*param*) *stmt*	*insert*(**id**.`name`, *param* .`kind`)
param	→	*type* **id**	
param	→	**ref** *type* **id**	
param	→	**result** *type* **id**	
call	→	**id** (*exp*)	*call* .`ok` =
exp	→	**id**	
exp_1	→	**id** [exp_2]	
exp_1	→	exp_2 *aritop* exp_3	

□

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.

$$\begin{array}{lcl} S & \to & TS \\ S & \to & T \\ T & \to & \#\, T \\ T & \to & \mathbf{a} \end{array}$$

(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 $S' \to S$. 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 "$\mathbf{a} \# \mathbf{a}$" 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). □

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 "`...`".

```
1 class Node { ... }
2 class Edge { ... }
3
4 class Graph {
5   Node startNode;
6   void connect(Node n1, n2) {
7     ... // connects two Nodes by creating an Edge-object ...
8   };
9 }
```

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.

```
1  class City extends Node {
2      String name;
3      ...
4  }
5
6  class Road extends Edge {
7      String name;
8      int distance;
9      ...
10 }
11
12
13 class RoadAndCityGraph extends Graph {
14     String country;
15     void connect(Node n1, n2) {
```

```
16          ... // connects to city objects treates as Nodes,
17              // by creating a Road object
18      };
19      void display () {
20          ... // display Roads and City with names
21      }
22  }
23
24  class TravelingSalesmanGraph extends RoadAndCityGraph {
25      void display () {
26          ... // display cities with names and roads
27              // with name and distance
28      };
29  }
```

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 $\langle classname \rangle :: \langle methodname \rangle$ 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`.

```
1   class Node { ... }
2   class Edge { ... }
3
4   class Graph {
5     Node startNode;
6     void connect(Node n1, n2) {
7       ... // connects two Nodes by creating an Edge-object ...
8     };
9   }
10
11
12
13  class RoadAndCityGraph extends Graph {
14      class City {
15          String name;
16          ...
17      }
18
19      class Road {
20          String name;
21          int distance;
22          ...
23      }
24
25      String country;
26      void connect(Node n1, n2) {
27          ... // connects to city objects treates as Nodes,
28              // by creating a Road object
29      };
30      void display () {
31          ... // display Roads and City with names
32      }
33  }
34
35
36
37
38
39
40  class TravelingSalesmanGraph extends RoadAndCityGraph {
41      void display () {
42          ... // display cities with names and roads
43              // with name and distance
44      };
```

```
45 }
```

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 allowed 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 instruction, which 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

```
1  a := input
2  b := input
3  d := a + b
4  c := a * b    // <- looky here
5  if ( b < 5) {
6    while (b < 0 ) {
7      a := b + 2
8      b := b + 1
9    }
10   d := 2 * b
11 } else {
12   d := b * 3
13   a := d - b
14 }
15 output a
16 output b
```

Which variables are *live* immediately at the end of line 4. Give a short explanation of your answer.

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.

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

```
1   t1 := a - b
2   t2 := b - c
```

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

Listing 3: 2AC

```
1 MOV  a, R0
2 MOV  b, R1
3 SUB R1, R0
4 SUB  c, R1
```

Ellen disagrees. Who is right? Explain your answer.

6 2011

Exercise 10 (CFGs and Parsing (25%))

Given are the following *3 separate grammars:*

$$A \rightarrow \mathbf{b}A\mathbf{c} \mid \boldsymbol{\epsilon} \quad (1)$$
$$A \rightarrow \mathbf{b}A\mathbf{b} \mid \mathbf{b} \quad (2)$$
$$A \rightarrow \mathbf{b}A\mathbf{b} \mid \mathbf{c} \quad (3)$$

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]

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.[3] Consider the following classed, defined in that assumed language

```
1  class A {
2    virtual void m (int x, y) { ... }
3    void p () { ... }
4    virtual void q() { ... }
5  }
6
7  class B extends A {
8    redef void m (int x, y) { ... }
9    void r() { ... }
10 }
```

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

```
11
12 class C extends A {
13   redef void q() { ... }
14 }
15
16 class D extends B {
17   redef void m (int x, y) { ... }
18 }
19
20 class E extend B {
21   redef void q() { ... }
22 }
23
24 class F extends C {
25   redef void m(int x, y) { ... }
26 }
```

(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 1. To indicate methods, use the same notation as before.

□

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.

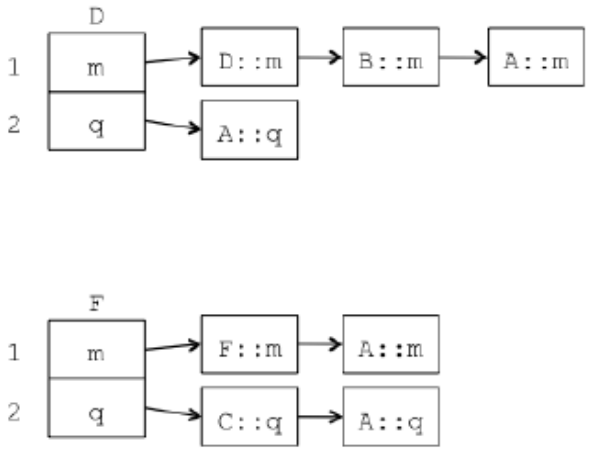


Figure 1: Extended virtual tables for `B` and `C`

class	→	**class name implements** *interfaces* **{** *decls* **}**
decls	→	*decls* **;** *decl* | *decl*
decl	→	*variable-decl* | *method-decl*
method-decl	→	*type* **name (** *params* **)** *body*
type	→	**int** | **bool** | **void**
interfaces	→	*interfaces* **,** *interface* | *interface*
interface	→	**name**

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

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

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***

□

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)

(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

`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 1: P-code instructions

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.

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). □

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.

```
1  {
2      class C {
3          void m1 () {
4              void f() {};
5
6              f ();
7          }
8          void m2 () {
9              int i;
10             void g() {
11                 int j;
12                 j = i;
```

```
13                  };
14                  i = 1;
15                  rC.m1();
16              };
17          };
18
19          C rc ;
20          void main () {
21              rc =  new C{} ; rC.m2{};
22          }
23      }
```

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). □

8 2013

Exercise 16 (CFG and parsing (35%))

Consider the following 2 grammars G_1 and G_2:

$$S \rightarrow \mathbf{(}\ S\ \mathbf{)} \mid \boldsymbol{\epsilon}$$

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

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:

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

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.

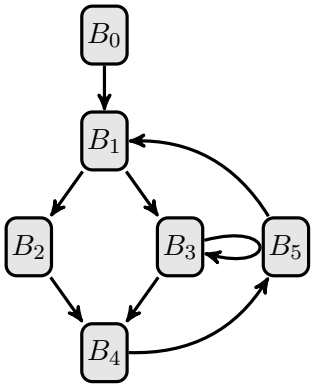
Exercise 17 (Code generation and analysis (25%))

(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:

```
1 ...
2 k = j + x
3 k = k * k
4 ...
```

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

```
1 ...
2 k' = j  + x    // k': new variable
3 k' = k' * k'
4 ...
```

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:

```
1 e = a - b
2 f = a - c
3 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.

□

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.