Universitetet i Oslo
Institutt for Informatikk
Reliable Systems
Martin Steffen, Gianluca Turin

INF 5110: Compiler construction

Spring 2021 **Series 8** 11. 05. 2021

Topic: (Intermediate) code generation (collection of exam questions)

Issued: 11. 05. 2021

Most of the exercises here are from earlier (written) exams. Some quite some years ago. They are also incorporated in the exam collection.

The first exercise here is *not* from an exam, but it's more of a recap anyway. The remainder are from the exams 2007, 2009, 2010, 2011, 2013, 2016. There might be even more years with questions about intermediate code generation or code generation. No need to include them all in this sheet, one can look at them in the exam collections.

Except 2016, the exams were not formulated by me. I somehow got hold of the exams, partly via students who had copies, and by previous lecturers. The older exams were mostly in Norwegian, so the text for those qis *not* the official exam text (but the meaning is preserved). I sometimes don't have access to official solutions in those years (of the exams resp. the corresponding exercises). Neither do I know how it was graded, for instance to which degree alternative formulations gave points, nor did I myself try to find all possible ways one could halfway correctly give an answer.

Actually, in some cases, I don't give a solution at all, because I have not solved them yet. Not because they are too complicated, simply because I have not found the time to work out solutions to all past generations of exams.

The purpose of the exercises (and going through old exams) is anyay not that I solve them, but that the participants try their knowledge and gain experience by doing them themselves.

Exercise 1 (Code generation) In this exercise we look at the code generation from the *notat* (i.e., from (Aho et al., 1986, page 538. . .))

(a) This is mean as repetition from the lecture. In the section for code generation, there is an example for which we showed at the very end of the section the resulting machine code. Look at the details how this algo generates this sequence. Try to determine a code sequence which is better (but does the same) than the one from that example. For the code, see the slide with the title *Code sequence* at the end.

(b) Discuss possibilities how one could improve the given algorithm from the lecture (taken from that book/notat).

(c) Translate the TAIC from Listing 1 to machine code using the algo from the notat/lecture. Consider some variations and improvements discussed in the previous point.

Assume that

- there are two registers initially "empty" and
- assume that for division "/", both source and destination have to reside in registers.

Listing 1: 3AIC

```
t := a − c
u := a + c
w := a / t
d := w + u
```

Exercise 2 (Code generation (-%))

(a) Given is the program from Listing 2. 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 2: 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.

Exercise 3 (Code generation (%))

Consider the following program in 3-address intermediate code.

Listing 3: 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 temporary 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.

Exercise 4 (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.

Exercise 5 (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.

`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

(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 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.

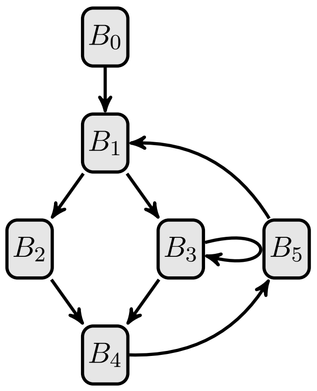
Exercise 6 (Code generation and analysis (25%))

(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 (if any) 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 they 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-instructions 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.

□

Exercise 7 (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.

(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. □

References

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