Universitetet i Oslo
Institutt for Informatikk

Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2024 — **Series 8** — 29. 4. 2024

Topic: Code generation

Issued: 29. 4. 2024

The exercises here are mostly taken from older exams. More exam questions, including more questions involving code generation are uploaded in a document `examcollection.pdf` (in the directory `handouts`).

Table 1 shows from which exams the exercises are taken. In many cases, I don't have solutions older exams.

Exercise 2	exam 2007, problem 4(a)
Exercise 3	exam 2009, problem 4
Exercise 4	exam 2010, problem 4(d)
Exercise 5 (3) + (4)	exam 2011, problem 4(c)+4(d)
Exercise 6	exam 2016, 4
Exercise 7	exam 2017, 6

Table 1: Exam questions

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...])

1. This is meant 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.

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

3. 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

```
1 t := a - c
2 u := a + c
3 w := a / t
4 d := w + u
```

Exercise 2 (Code generation (-%))

1. Given is the program from Listing 4. 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
```

1. 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.

2. 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.

3. 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.

4. 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.

5. 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 (–%))

1. 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 4: 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.

`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 2: P-code instructions

Exercise 5 (Code generation & P-code (25%))

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

   (a) 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.

   (b) Sketch which *data structures*

2.

3. 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 whether the user-level source language assures an *order* of evaluation of compound expressions. Other factors you think relevant can be discussed as well.

4. 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 (%))

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*).

1. Register descriptors indicate, for each register, which variables have their value in this register.

   (a) 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.

2. 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. □

Exercise 7 (Code generation (%))

(a) Consider the following transformation on three-address code, illustrated on the following example.

Listing 5: Before

```
1 if t == 0
2 then
3    x = y + z;
4    <rest of then-branch>
5 else
6    x = y + z;
7    <rest of else-branch>
8 endif
```

Listing 6: After

```
1 x = y + z;
2 if t == 0
3 then
4    <rest of then-branch>
5 else
6    <rest of else-branch>
7 endif
```

The idea is to move "common instructions" (like the assignment `x = y + z` in the example) *before* the conditional, so long it does not change the semantics of the code. The three address code in this sub-problem supports two-armed conditionals (if-then-else), not the if-goto constract as in the lecture and in sub-task (b).

Assume code generation as covered in the "notat" which covers parts of Chapter 9 of the old "dragon book" (*Compilers: Principles, Techniques, and Tools, A. V. Aho, R.*

Sethi, and J. D. Ullman, 1986). Assume further that the code generator has access to *local* liveness information, i.e., liveness information per basic block, but *no global liveness* information is available.

Under these assumptions, what are potential effects of the code transformation on the qualitity of the generated code? Discuss this question referring to the cost model of the notat/lecture.

Note: neither exact sequences of possibly generated two-address code nor detailed calculations of costs are expected/needed as answer, just a short discussion of the influence of the transformation on factors of the cost model.

(b) Consider the program from Listing 13 in three address code. We do not distinguish here between temporaries and standard variables.

(a) Indicate the *basic blocks* in giving start and end line for each block (numbering the blocks like B_0, B_1, etc.) You can also use the code repeated in the appendix, drawing clearly visible horizontal lines indicating the boundaries of the blocks and give the B_i-numbers of the blocks.

(b) Draw the control flow graph of the program using B_0, B_1 from the previous question to identify the nodes of the graph.

(c) Does the control-flow graph contain a *loop?* Use the notion of loops for control-flow graphs from the lecture.

(d) Give the *inLive* and *outLive* information for each block (best in the form of a table).

Listing 7: Three-address code

```
1  x = input
2  y = input
3  label L1
4  b = x + y
5  z = b*z
6  label L2
7  x = a + 1
8  if_false x goto L3
9  x = y + x
10 if_true z goto L5
11 goto L1
12 label L3
13 z = b * 2
14 goto L2
15 label L5
16 output x
```

References

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