Linköpings universitet
IDA Department of Computer and Information Sciences
Prof. Dr. Christoph Kessler

TENTAMEN / *EXAM*

TDDB29 Kompilatorer och interpretatorer / *Compilers and interpreters*
TDDB44 Kompilatorkonstruktion / *Compiler construction*

26 mar 2008, 08:00–12:00

Jour: 08:00–12:00 Christoph Kessler, 070-3666687, 013-282406

Hjälpmedel / *Admitted material:*

– Engelsk ordbok / *Dictionary from English to your native language*;
– Miniräknare / *Pocket calculator*

General instructions

- This exam has 9 assignments and 5 pages, including this one.
  Read all assignments carefully and completely before you begin.
- The first assignment (on formal languages and automata theory) is ONLY for TDDB29, while the last one (on code generation for RISC...) is ONLY for TDDB44.
- It is recommended that you use a new sheet for each assignment. Number all your sheets, and mark each sheet on top with your name, personnummer, and the course code.
- You may answer in either English or Swedish.
- Write clearly. Unreadable text will be ignored.
- Be precise in your statements. Unprecise formulations may lead to a reduction of points.
- Motivate clearly all statements and reasoning.
- Explain calculations and solution procedures.
- The assignments are *not* ordered according to difficulty.
- The exam is designed for 40 points (per course). You may thus plan about 5 minutes per point.
- Grading: U, 3, 4, 5. For exchange students (with a *P* in the personnummer) ECTS marks will be applied.

  The preliminary threshold for passing (grade 3) is 20 points.
- **OBS C:are antagna före 2001:** Om du vill ha ditt betyg i det gamla betygsystemet (U, G, VG) skriv detta tydligt på omslaget av tentan. Annars kommer vi att använda det nya systemet (U, 3, 4, 5).

1. **Only TDDB29:** (6 p.) **Formal languages and automata theory**

   Consider the language L consisting of all strings w over the alphabet $\{0, 1\}$ such that if w ends with 11 then it must contain an even number of ones.

   (a) Construct a regular expression for L. (1.5p)

   (b) Construct from the regular expression an NFA recognizing L. (1.5p)

   (c) Construct a DFA recognizing L, either by deriving from the NFA of question (1b), or by constructing one directly. (2.5p)

   (d) Give an example of a formal language that is context-free but cannot be recognized by a finite automaton. (0.5p)

2. (6p) **Compiler structure**

   (a) What are the advantages and disadvantages of a multi-pass compiler (compared to a one-pass compiler)? (1p)

   (b) Describe what phases normally are found in a compiler, what is their purpose, how they are connected, and what is their input and output. (3p)

   (c) Most modern compilers have not just one but several intermediate representations.

      i. Explain how these are, in general, related to each other, and what this organization means for the code generation process. (1p)

      ii. What is the main advantage of having more than one IR, and what could be a drawback? (1 p)

3. (6p) **Top-Down Parsing**

   (a) Given a grammar with nonterminals $\langle Expr \rangle$, $\langle SimpExpr \rangle$, $\langle ArrayRef \rangle$ and $\langle IndexExprs \rangle$ and the following productions:

   $$\langle Expr \rangle \rightarrow \langle SimpExpr \rangle \mid \langle ArrayRef \rangle$$
   $$\langle SimpExpr \rangle \rightarrow \textbf{id}$$
   $$\langle ArrayRef \rangle \rightarrow \textbf{id} \; [\; \langle IndexExprs \rangle \;]$$
   $$\langle IndexExprs \rangle \rightarrow \langle IndexExprs \rangle \; , \; \langle SimpExpr \rangle \mid \langle SimpExpr \rangle$$

   where $\langle Expr \rangle$ is the start symbol.

   What is/are the problem(s) with this grammar if it is to be used for writing a recursive descent parser with a single token lookahead? Resolve the problem(s), and write a recursive descent parser for the modified grammar. *(Pseudocode is fine. Use function* `scan()` *to read the next input token.)* (4.5p)

   (b) Sketch the main idea of panic mode error recovery in a predictive (LL) parser. (1.5p)

4. (6 p.) **LR parsing**

   Given the following grammar G for strings over the alphabet $\{x, y, z, t\}$, with nonterminals A, B and C, where A is the start symbol:

$$
\begin{aligned}
A &::= x\ B\ x \mid C \\
B &::= y\ t\ B \mid y \\
C &::= x\ C\ z \mid y
\end{aligned}
$$

   Is the grammar G in SLR(1)? Is it LR(0)? Motivate with the LR-item sets.

   Construct the characteristic LR-item NFA, the corresponding GOTO graph, the ACTION table and the GOTO table.

   Show with tables and stack how the string $xytyx$ is parsed.

5. (6 p.) **Syntax-directed translation**

   The following grammar rule describes a for loop

$$\langle loop \rangle \ ::= \ \texttt{for}\ \langle id \rangle\ \texttt{in}\ \langle expr \rangle..\langle expr \rangle\ \texttt{do}\ \langle stmt\_list \rangle\ \texttt{enddo}$$

   where loop variable $\langle id \rangle$ will, at run time, take the values between the two expressions (the endpoints included) one by one, and for each value, $\langle stmt\_list \rangle$ is executed. For example,

```
for i in 2..7 do
   print(i);
enddo
```

   prints the numbers 2 to 7.

   Write a syntax-directed translation scheme, using attributes and semantic rules, for the grammar rule above. The values of the two $\langle expr \rangle$ are computed in the beginning and cannot be changed during the execution of the loop. Neither can the loop variable $\langle id \rangle$ be changed inside the loop.

   You may either use symbolic labels (generated by *newlabel()*) or work directly on quadruple numbers, using backpatching if necessary, but be consistent. Temporary variables can be generated by *gentemp()*.

6. (4.5 p.) **Intermediate code generation**

   Translate the following code segment into abstract syntax tree, quadruples, and postfix code: (4.5 p)

```
x = 17;
y = 3;
while (x>y) {
  x = x+1;
  if (y<20)
    y = 3*(x-y);
  else
    y = 2*y;
  print(y);
}
```

7. (3 p.) **Simple code generation**

   Given the following basic block given in pseudo-quadruple form:

```
1:  T1 := - e
2:  T2 := a + b
3:  T3 := c - d
4:  T4 := T2 * T3
5:  f := T1 / T4
```

   (a) Draw the data flow graph of the basic block. Assume a register machine with load-store architecture, i.e., arithmetic operations can only work on registers. Assume that, initially, the variables `a`, `b` etc. read by the program are not in registers and thus must be loaded into registers as appropriate. At the end, the value of variable `f` must be stored to memory. Expose the memory accesses as nodes in the data flow graph. (1p)

   (b) Use the *Labeling algorithm* to compute labels for the dataflow graph nodes, and generate code (pseudo-instructions, one per arithmetic or memory operation) for this basic block such that the program variables and the temporaries `T`i are properly assigned to registers and the code uses a *minimum* number of registers. Assume that each instruction executes in one clock cycle and that an operand register could be reused to store the result of an instruction. How many registers do you need? (2p)

8. (2.5 p.) **Target code generation**

   Explain the idea of instruction selection for expression trees by tree pattern matching. (Be general! An example is nice but not sufficient.)

   Why is this technique more powerful than simple macro expansion, and what kind of processor architectures will profit most from this technique? (2.5p)

9. **Only TDDB44:** (6 p.) **Code generation for RISC ...**

   (a) A *structural hazard* in a pipelined processor is a resource conflict where several instructions compete for the same hardware resource in the same clock cycle.

      i. How do superscalar processors handle structural hazards? (0.5p)

      ii. Certain processor architectures leave it to the assembler-level programmer or compiler to make sure that structural hazards do not occur. Sketch the technique (data structure, principle) that can be used in instruction scheduling to avoid structural hazards. (1p)

   (b) Given the following medium-level intermediate representation of a program fragment (derived from a `while` loop):

```
1:   c = 3
2:   e = 1.0
3:   goto 9
4:   a = c / 2
5:   b = a + e
6:   c = a - b
7:   d = e
8:   e = e * 0.5
9:   f = (e > 0.1)
10:  if f goto 4
11:  d = d / b
```

   Identify the live ranges of program variables, and draw the live range interference graph

   (i) for the loop body in lines 4–9,

   (ii) for the entire fragment.

   For both (i) and (ii), assign registers to all live ranges by coloring the live range interference graph. How many registers do you need at least, and why? (3.5p)

   (c) Register allocation and instruction scheduling are often performed separately (in different phases). Explain the advantages and problems of this separation. (1p)

Good luck!