Linköpings universitet
IDA Department of Computer and Information Sciences
Prof. Peter Fritzson and Doc. Christoph Kessler

TENTAMEN / *EXAM*

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

19 apr 2006, 08:00–12:00

Jour: 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 4 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. The preliminary threshold for 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 $\{a, b\}$ such that if w contains at least 3 a's (not necessarily in sequence) then it must contain exactly two b's.

   (a) Construct a regular expression for L. (1p)

   (b) Construct from the regular expression an NFA recognizing L. (1p)

   (c) Construct a DFA recognizing L, either by deriving from the NFA of question (1b), or by constructing one directly. (3p)

   (d) We have learned that "finite automata cannot count". Give an example of a formal language that cannot be recognized by a finite automaton. (0.5p)

   Yet, for the language L above there exists a finite automaton recognizing it. Why is this not a counterexample? (0.5p)

2. (4p) **Phases and passes**

   (a) What are the advantages and disadvantages of a multi-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)

3. (5p) **Top-Down Parsing**

   (a) Given a grammar with nonterminals S and X and the following productions:

   $S \ \rightarrow \ aS \mid aX$

   $X \ \rightarrow \ XbX \mid d$

   where S 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) The theory for formal languages and automata says that a stack is required for being able to parse context-free languages. We have used such a stack, for instance, in the LL-item pushdown automaton in the lecture on top-down parsing. But where is the corresponding stack in a recursive descent parser? (0.5p)

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

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

$$
\begin{aligned}
A &::= xAyAz \mid xBzAx \mid \epsilon \\
B &::= yBzBx \mid yBxBz \mid \epsilon
\end{aligned}
$$

   If G is SLR(1) or even LR(0), construct the canonical LR-items and the LR-item DFA for the grammar and show with tables and stack how the string `xyz` is parsed. *If* G is *not* SLR(1) or LR(0), then explain why.

5. (3 p.) **Memory management**

   What is an activation record? What properties of a programming language lead to a need for activation records? What does an activation record contain?

6. (7 p.) **Intermediate code generation**

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

   ```
   x = 123;
   y = 3;
   while (x>100) {
     x = x - y;
     y = 2*y;
   }
   ```

   (b) Given the following program fragment in pseudo-quadruple form:

   ```
   1:  T1 := a + b
   2:  T2 := T1 - c
   3:  x  := T2
   4:  T3 := x > 0
   5:  if T3 goto 10
   6:  T4 := x < 0
   7:  if T4 goto 1
   8:  T5 := a + b
   9:  goto 4
   10: m := T5
   ```

   Divide this program fragment into *basic blocks* and then draw the *control flow graph* for the program fragment. (2.5p)

7. (6 p.) **Syntax-directed translation**

   An Algol-like language is augmented with an `if2` statement in the following way:

   ```
   <if2_statement> ::= if2(<expression_1>,<expression_2>)
                        none: <statement_1>
                        first: <statement_2>
                        second: <statement_3>
                        both: <statement_4>
                       endif2;
   ```

   The `if2` statement works like the following nesting of `if` statements:

   ```
   if <expression_1>
    then if <expression_2>
         then <statement_4>
         else <statement_2>
   ```

```
else if <expression_2>
      then <statement_3>
      else <statement_1>;
```

Write the semantic rules — a syntax directed translation scheme — for translating the `if2` statement to quadruples. Assume that the translation scheme is to be used in a bottom-up parsing environment using a semantic stack. Use the grammar rule above as a starting point, but maybe it has to be changed.

You are not allowed to define and use symbolic labels, i.e., all jumps should have absolute quadruple addresses as their destinations. Explain all the attributes, functions, and instructions that you introduce. State all your assumptions.

8. (3 p.) **Bootstrapping**

   Explain the concepts of *rehosting* and *retargeting*. Use T-diagrams.

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

   (a) Explain the main similarity and the main difference between superscalar and VLIW architectures from a compiler's point of view. Which one is harder to generate code for, and why? (1.5p)

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

   ```
   1:   c = 3;
   2:   k = 20;
   3:   if k<=0 goto 9;
   4:   a = c / 2;
   5:   b = a + c;
   6:   c = a * b;
   7:   k = k - 1;
   8:   goto 3;
   9:   d = b * c
   ```

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

   (i) for the basic block in lines 4–7 (i.e., the loop body),

   (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!