Linköpings universitet
IDA – Department of Computer and Information Science
Professor Peter Fritzson

TENTAMEN / *EXAM*

TDDD16 Kompilatorer och interpretatorer / *Compilers and Interpreters*
TDDB29 Kompilatorer och interpretatorer / *Compilers and Interpreters*
TDDB44 Kompilatorkonstruktion / *Compiler Construction*

16 dec 2008, 08:00–12:00, TER1,TER2

Jour: Peter Fritzson, 0708-281484, 013-281484; (Will come approx 9.15-9.35)

Hjälpmedel / *Allowed material:*

- Engelsk ordbok / Dictionary from/to English to/from 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 TDDB16/29, while the last one (on code generation for RISC, etc.) 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, personal number/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 6 minutes per point.
- Grading: U, 3, 4, 5.
- For exchange students (with a in the personnummer) ECTS marks will be applied.
- The preliminary threshold for passing (grade 3) is 20 points.

1. Only TDDD16/TDDB29: (6p) Formal Languages and Automata Theory

Consider the language *L* consisting of all strings *w* over the alphabet {0,1} such that *w* contains 11 or 0000 (i.e., at least 2 ones in sequence, or at least 4 zeroes in sequence, or both). For example, the strings 01011, 000010, 1100001, 11, etc. belong to the language *L*.

(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 not context-free. (0.5p)

2. (6p) Compiler Structure and Generators

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

(b) Describe briefly what phases (at least 4 phases, max 5) normally are found in a compiler, what is their purpose, how they are connected, what is their input and output, and for each phase give an example of a formalism this phase may be generated from. (4p)

(c) How does a just-in-time (JIT) compiler work? (1p)

3. (5p) Top-Down Parsing

(a) Given a grammar with nonterminals `<Expr>`, `<SimpExpr>`, `<ArrayRef>` and `<IndexExprs>` and the following productions:

```
<Expr>       =>  Id | <ArrayRef>
<SimpExpr>   =>  Id  | Colon
<ArrayRef>   =>  Id [<IndexExprs>]
<IndexExprs> =>  <IndexExprs> , <SimpExpr> | <SimpExpr>
```

where `<Expr>` is the start symbol, `Id` and `Colon` are terminals. This may for example generate expressions such as `arr[c,:,:,d,:]`, where `arr` is an array name and `c` or `d` could be constant integer variables. The colon can be used to specify array slices.

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,w,z,t`}with nonterminals `A`, `B` and `C` where `A` is the start symbol:

```
A ::= x B x  | C
B ::= w t t B  | w
C ::= x C z  | w
```

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 `xwttwx` is parsed.

5. (3 p) Memory Management

What does an activation record contain? What happens on the stack at function call? What happens on the stack at function return?

6. (5 p) Syntax-Directed Translation

The REPEAT statement in a Pascal-like language could be described using this rule:

```
<repeatstmt> ::= REPEAT <stmt> UNTIL <expr>
```

The semantics of the REPEAT statement is that statement `<stmt>` is executed and then repeated as long as expression `<expr>` evaluates to zero.

Write the semantic rules — a syntax directed translation scheme — for translating the REPEAT 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.

In order to obtain full points, 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.

(For an otherwise correct solution that uses symbolic labels up to 3p can be given.)

7. (6 p) Intermediate Code Generation

Given the following code segment:

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

(a) Translate the code segment into abstract syntax tree and quadruples. (4 p)

(b) In the quadruple representation, identify the basic blocks of the program, and draw the basic block graph. (2p)

(If you cannot derive the quadruple representation, you may do it instead for the source code above (1p).)

8. (3 p) Simple Code Generation

The following basic block given in pseudo-quadruple form:

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

```
5: T5 := T4 * g
6: f := T1 / T5
```

(a) Draw the data flow graph of the basic block. Assume a register machine with loadstore 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` 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)

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: x = 3
2: y = 4.1
3: goto 9
4: a = x / 3
5: b = a + y
6: x = a - b
7: d = y
8: y = y * 0.5
9: f = (y > 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) What is the basic idea of *software pipelining* of loops? (1p)

Good luck!