Försättsblad till skriftlig tentamen vid Linköpings universitet

(fylls i av ansvarig)

Datum för tentamen	13-03-25
Sal	TER4
Tid	08-12
Kurskod	TDDD55
Provkod	TEN1
Kursnamn/benämning	Kompilatorer och interpretatorer
Institution	*IDA*
Antal uppgifter som ingår i tentamen	10
Antal sidor på tentamen (inkl. försättsbladet)	
Jour/Kursansvarig	Jonas Wallgren
Telefon under skrivtid	- (i tentasalens närhet)
Besöker salen ca kl.	
Kursadministratör (namn + tfnnr + mailadress)	Carita Lilja, 28 1463, carli@ida.liu.se
Tillåtna hjälpmedel	Se tentamens första sida
Övrigt (exempel när resultat kan ses på webben, betygsgränser, visning, övriga salar tentan går i m.m.)	
Vilken typ av papper ska användas, rutigt eller linjerat	
Antal exemplar i påsen	7

Tentamen/Exam

TDDB44 Kompilatorkonstruktion / Compiler Construction
TDDD55 Kompilatorer och interpretatorer /
Compilers and Interpreters

2013–03–25, 08.00 – 12.00

Hjälpmedel / Allowed material:

- Engelsk ordbok / Dictionary from/to English to/from your native language
- Miniräknare / Pocket calculator

General instructions:

- Read all assignments carefully and completely before you begin
- **Note that not every problem is for all courses.** Watch out for comments like "TDDD55 only".
- You may answer in Swedish or in English.
- Write clearly — unreadable text will be ignored. Be precise in your statements — unprecise formulations may lead to 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 6 minutes per point.
- Grading: U, 3, 4, 5 resp. Fx, C, B, A.
- The preliminary threshold for passing (grade 3/C) is 20 points.

1. (TDDD55 only - 6p) **Formal Languages and Automata Theory**
   Consider the language L consisting of all strings w over the alphabet $\{0, 1\}$ such that every string contains 00 once *or* 11 once. Example of strings in the language: 1001, 010<u>11</u>01, <u>11</u>0, 1<u>00</u>. Examples of strings *not* in the language: 0101, 1<u>00</u>10<u>11</u>, 010<u>11</u>0<u>11</u>, <u>000</u>. (The last example counts as two occurences of 00.)

   (a) (2p) Construct a regular expression for L.

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

   (c) (2.5p) Construct a DFA recognizing L, either by deriving it from the NFA or by constructing it directly.

2. (3p) **Compiler Structure and Generators**
   Describe briefly what phases are found in a compiler. What is their purpose, how are they connected, what is their input and output?

3. (2p) **Symbol Table Management**
   Describe what the compiler — using a symbol table implemented as a hash table with chaining and block scoped control — does in compiling a statically scoped, block structured language when it handles:

   (a) (0.5p) block entry

   (b) (0.5p) block exit

   (c) (0.5p) a variable declaration

   (d) (0.5p) a variable use.

4. (6p) **Memory management**

   (a) (1p) What property of programming languages requires the use of activation records?

   (b) (1p) What does an activation record contain?

   (c) (2p) What happens on the stack at function call and at function return?

   (d) (2p) What are static and dynamic links? How are they used?

5. (6p) **Top-Down Parsing**
Given a grammar with nonterminals S, P, Q, and R, where S is the start symbol, and the following productions:

```
1. S ::= P $ P
2. P ::= P £ Q
3.     | Q
4. Q ::= Q # R
5.     | R
6. R ::= 0
7.     | 1
```

Assume that $, £, and # are operators.

(a) (1p) What is the associativity (right, left, none) of the operators?

(b) (1p) What is the precedence (relative priority) between the operators?

(c) (2p) Can the grammar be used directly for a recursive-descent parser? Motivate your answer. If not, rewrite the grammar so that the language it defines can be parsed using the recursiuve-descent method.

(d) (2p) Write a recursive-descent parser to analyze the language defined by the grammar.

6. (TDDD55 only - 6p) **LR parsing**

(a) (3p) Use the SLR(1) tables below to show how the string a^b*a+b is parsed. You should show, step by step, how stack, input data etc. are changed during the parsing. Start state is 00, start symbol is S.

Grammar:

```
1. S ::= X + X
2. X ::= Y * X
3.     | Y
4. Y ::= Y ^ Z
5.     | Z
6. Z ::= a
7.     | b
```

Tables:

	Action						GOTO			
State	$	+	*	^	a	b	S	X	Y	Z
00	*	*	*	*	S11	S10	05	04	08	12
01	*	*	*	*	S11	S10	*	06	08	12
02	*	*	*	*	S11	S10	*	07	08	12
03	*	*	*	*	S11	S10	*	*	*	09
04	*	S01	*	*	*	*	*	*	*	*
05	A	*	*	*	*	*	*	*	*	*
06	R1	*	*	*	*	*	*	*	*	*
07	R2	R2	*	*	*	*	*	*	*	*
08	R3	R3	S02	S03	*	*	*	*	*	*
09	R4	R4	R4	R4	*	*	*	*	*	*
10	R7	R7	R7	R7	*	*	*	*	*	*
11	R6	R6	R6	R6	*	*	*	*	*	*
12	R5	R5	R5	R5	*	*	*	*	*	*

(b) (3p) Explain the concept of conflict in LR parsing — what it is, how it could be handled.

7. (TDDB44 only - 6p) **LR parsing**
Given the following grammar G for strings over the alphabet a,b,+,*,(,)with nonterminals E, T, and F where E is the start symbol:

```
E ::= T | E*T
T ::= F | T+F
F ::= a | b |(E)
```

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 `a+b*b+(a*a)` is parsed.

8. (5p) **Syntax-Directed Translation**
A loop statement that combines pre-test and post-test could be described like:

```
<doubletestloop> ::= WHILE <expr> DO <stmt> UNTIL <expr>;
```

If the first `<expr>` evaluates to true then the statement `<stmt>` is executed. If the second `<expr>` then doesn't evaluate to true the whole `<doubletestloop>` is executed again.

Write a syntax-directed translation scheme, with attributes and semantic rules, for translating the `<doubletest>` 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 it maybe 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.

9. (6p) **Intermediate Code Generation**
Given the following code segment in a Pascal-like language:

```
if x<y
   then while x>z
             x:=x-10
   else y:=factorial(fibonacci(x)+1);
```

   (a) (3p) Translate the code segment into an abtract syntax tree, quadruples, and postfix code.

   (b) (3p) Divide the following code inte basic blocks, draw a control flow graph, and show as well as motivate the existing loop(s).

```
       goto L2
   L1: x:=x+1
   L2: x:=x+1
   L3: x:=x+1
       if x=1 then goto L1
       if x=2 then goto L4
       goto L5
   L4: x:=x+1
   L5: x:=x+1
       if x=4 then goto L3
```

10. (TDDB44 only - 6p) **Code Generation for RISC etc.**

   (a) (2p) Explain the main characteristics of CISC and RISC architectures, and their differences.

   (b) (1.5p) Explain what register allocation and register assignment is (in the context of code generation), and what the difference is.

   (c) (1.5p) Explain briefly the concept of software pipelining. Show it with a simple example.

   (d) (1p) What is a live range? Explain the concept and show a simple example.