Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2017-01-14
Sal (2)	U6(16) U7(2)
Tid	8-12
Kurskod	TDDD55
Provkod	TEN1
Kursnamn/benämning **Provnamn/benämning**	Kompilatorer och interpretatorer En skriftlig tentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	12
Jour/Kursansvarig Ange vem som besöker salen	Martin Sjölund
Telefon under skrivtiden	070 - 7567358

Tentamen/Exam

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

2017–01–14, 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 01 once *or* 10 once *but not both.* Example of strings in the language: 111000, 0111. Examples of strings *not* in the language: 01010, 111.

   (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. (4p) **Compiler Structure and Generators**

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

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

   (a) (4.5p) Given a grammar with nonterminals S, A, B and the following productions:

```
S ::= S u | P Q v | P Q w
P ::= P x | y
Q ::= Q z | e
```

   where `S` is the start symbol, `u, v, w, x, y, z` are terminals. (`e` is the empty string!) 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/program code without declarations is fine. Use the function scan() to read the next input token, and the function error() to report errors if needed.)

   (b) (0.5p) 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?

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

   (a) (3p) Use the SLR(1) tables below to show how the string `x+y*x-y` 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 ::= P * P
2. P ::= Q + P
3.     | Q
4. Q ::= Q - R
5.     | R
6. R ::= x
7.     | y
```

Tables:

	Action						Goto			
State	$	*	+	-	x	y	S	P	Q	R
00	*	*	*	*	S09	S10	01	02	05	08
01	A	*	*	*	*	*	*	*	*	*
02	*	S03	*	*	*	*	*	*	*	*
03	*	*	*	*	S09	S10	*	04	05	08
04	R1	*	*	*	*	*	*	*	*	*
05	R3	R3	S06	S11	*	*	*	*	*	*
06	*	*	*	*	S09	S10	*	07	05	08
07	R2	R2	*	*	*	*	*	*	*	*
08	R5	R5	R5	R5	*	*	*	*	*	*
09	R6	R6	R6	R6	*	*	*	*	*	*
10	R7	R7	R7	R7	*	*	*	*	*	*
11	*	*	*	*	S09	S10	*	*	*	12
12	R4	R4	R4	R4	*	*	*	*	*	*

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

6. (TDDB44 only - 6p) **LR parsing**
Given the following grammar G for strings over the alphabet {a,b,c,d} with nonterminals X and Y, where X is the start symbol:

```
X ::= Xb | aX | aYb | c
Y ::= bY | Ya | bXa | d
```

Is the grammar G in SLR(1) or even LR(0)? Justify your answer using the LR item sets.
If it is: construct the characteristic LR-items NFA, the corresponding GOTO graph, the ACTION table and the GOTO table and show with tables and stack how the string abcab is parsed.
If it is not: describe where/how the problem occurs.

7. (5p) **Syntax-Directed Translation**
A Pascal-like language is extended with a `signselect` statement according to the following grammar:

```
<signselect> ::= signselect <expression>
                        neg: <statement>
                       zero: <statement>
                        pos: <statement>
                    endselect
<statement> ::= <assignment> | ... | <signselect>
```

(where "..." represents all other possible kinds of statements). First, `<expression>` is evaluated. Then, depending on the resulting value being less than, equal to, or larger than zero the corresponding statement is executed. Only one of the `<statement>` parts is executed. Then execution continues after the `signselect` statement.

Example:

```
signselect 3-15+2
      neg: print("This one")
     zero: print("Not this one")
      pos: print("Not this one")
 endselect;
```

Write a syntax-directed translation scheme, with attributes and semantic rules, for translating `signselect` statements to quadruples. The translation scheme should be used during bottom-up parsing. You are not allowed to define and use symbolic labels, i.e. all jumps should have absolute quadruple addresses as their destinations. You may need to rewrite the grammar. Explain all the attributes, functions, and instructions that you introduce. State all your assumptions. (Since it is a syntax-directed translation scheme, not an attribute grammar, generation of a quadruple puts it in an array of quadruples and attribute values are "small" values such as single quadruple addresses.)

8. (3p) **Error Handling**
   Explain, define, and give examples of using the following concepts regarding error handling:

   (a) (1p) Valid prefix property,

   (b) (1p) Phrase level recovery,

   (c) (1p) Global correction.

9. (3p) **Memory management**

   (a) (1p) Non-local references: How does a static link work?

   (b) (1p) Non-local references: How does a display work?

   (c) (1p) Dynamic data: How is the actual size and contents of a dynamic array handled?

10. (6p) **Intermediate Code Generation**

    (a) (3p) Given the following code segment in a Pascal-like language:

```
if fac(x)<65536
   then repeat
          y=x+4711
        until x<fib(y)
   else z=2
```

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

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

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

(b) (1p) Explain briefly the concept of software pipelining.

(c) (3p) Given the following medium-level intermediate representation of a program fragment:

```
1: a = 1.0
2: b = 1.0
3: c = 3.0
4: e = 2.0
5: goto 9
6: b = a + b
7: a = c / 2.0
8: c = a * e
9: e = e / 2.0
10: f = (e > 0.1)
11: if f goto 6
12: d = 1.5 * a
```

Identify the live ranges of program variables, and draw the live range interference graph for the entire fragment. Assign registers to all live ranges by coloring the live range interference graph. How many registers do you need at least, and why?

12. (TDDB44 only - 3p) **Compiler Lab Exercises**
Correct and complete labs from the 2016 TDDB44 lab course handed in at the latest December 21, 2015, will give 3 points. State if you think that you have fulfilled the conditions and you should receive these points.