Information page for written examinations at Linköping University

Examination date	2019-04-26
Room (1)	TER2(1)
Time	14-18
Edu. code	TDDD55
Module	TEN1
Edu. code name **Module name**	Compilers and Interpreters (Kompilatorer och interpretatorer) Written examination (En skriftlig tentamen)
Department	IDA
Number of questions in the examination	10
Teacher responsible/contact person during the exam time	Martin Sjölund
Contact number during the exam time	+46707567358
Visit to the examination room approximately	15:30
Name and contact details to the course administrator (name + phone nr + mail)	Veronica Kindeland Gunnarsson 013-28 56 34
Equipment permitted	English dictionary Pocket calculator
Other important information	
Number of exams in the bag	

Tentamen/Exam

TDDD55 Kompilatorer och interpretatorer / Compilers and Interpreters

2019-04-26, 14:00 – 18:00

Hjälpmedel / Allowed material:

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

General instructions:

- Read the instructions and examination procedures for exams at LiU.
- Read all assignments carefully and completely before you begin.
- You may answer in Swedish or in English.
- Write clearly – unreadable text will be ignored. Be precise in your statements – imprecise formulations may lead to reduction of points. Motivate clearly all statements and reasoning. Explain calculations and solution procedures.
- The assignments are sorted into fundamentals (20p) and an advanced section (max 10p).
  - Solve all of the fundamental assignments.
  - Choose up to 10p worth of the advanced assignments (there are 6 assignments to choose from). You are penalized if you answer more than 10p worth of advanced assignments.
- The exam is designed for 30 points in 240 minutes. You may thus plan 8 minutes per point.
- Grading: U, 3, 4, 5 resp. Fx, C, B, A.
- The preliminary threshold for passing (grade 3/C) is 15 points, of which 10 points should be from the fundamentals section.

Fundamentals

1. (6p) **Formal Languages and Automata Theory**

   Consider the language L consisting of all strings w over the alphabet {`a`, `b`, `c`} such that a string is accepted if it contains the substring `ab` but not the substring `cb`. Example of strings in the language: `aabacccc`, `ccabcab`. Examples of strings not in the language: `cbabc`, `abcba`.

   (a) (1.5p) 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.

   (d) (0.5p) Give an example of a formal language that is not context-free.

2. (3p) **Compiler Structure and Generators**

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

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

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

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

   ```
   S ::= S 1
   S ::= X Y 2
   S ::= X Y 6
   X ::= X 3
   X ::= 4
   Y ::= X 5
   Y ::= ε
   ```

   where S is the start symbol, 1, 2, 3, 4, 5 and 6 are terminals. (ϵ 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?

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

Use the SLR(1) tables below to show how the string 1214341 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 ::= A
2. A ::= 1 A 2
3.     | 1 B 1
4.     | 1 B 3
5. B ::= 1 A 2
6.     | 2 A 4
7.     | 4
```

Tables:

	Action					GOTO		
State	$	1	2	3	4	S	A	B
00	*	S02	*	*	*	*	01	*
01	A	*	*	*	*	*	*	*
02	*	S08	S12	*	S11	*	03	05
03	*	*	S04	*	*	*	*	*
04	R2	*	R2	*	R2	*	*	*
05	*	S06	*	S07	*	*	*	*
06	R3	*	R3	*	R3	*	*	*
07	R4	*	R4	*	R4	*	*	*
08	*	S08	S12	*	S11	*	09	05
09	*	*	S10	*	*	*	*	*
10	R2	R5	R2	R5	R2	*	*	*
11	*	R7	*	R7	*	*	*	*
12	*	S02	*	*	*	*	13	*
13	*	*	*	*	S14	*	*	*
14	*	R6	*	R6	*	*	*	*

Advanced

Read the general instructions first. You should not answer all of these assignments.

5. (3p) **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) block entry

   (b) block exit

   (c) a variable declaration

   (d) a variable use.

6. (5p) **Syntax-Directed Translation**

   A Pascal-like language is extended with a `restartblock` statement according to the following grammar:

```
<block>     ::= begin <stmt_list> end
<stmt_list> ::= <stmt_list><stmt> |
<stmt>      ::= <assignment> | ... | restartblock
```

   (where "..." represents all other possible kinds of statements). `restartblock` means that execution restarts at the beginning of the immediately enclosing block.

   Example:

```
begin
    x:=17;
L1: begin
        y:=y-42;
        if p=4711
L2:         then restartblock;
            else q:=q-1;
L3: end;
end;
```

   where `restartblock` at `L2` means a jump to `L1` (i.e. the beginning of the enclosing block).

   (a) (4p) Write a syntax-directed translation scheme, with attributes and semantic rules, for translating `<block>`s, and `restartblock`s inside them, 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.)

(b) (1p) What problem would occur in handling of the translation scheme if instead of `restartblock` there would be an `exitblock` statement that jumped to the `end` of the immidiately enclosing block (instead of the `begin`), i.e. to `L3` in this example?

7. (1p) **Error Handling**

   Explain, define, and give examples of using the valid prefix property regarding error handling.

8. (2p) **Memory management**

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

   (b) (1p) What are static and dynamic links? How are they used?

9. (3p) **Intermediate Representation**

   Given the following code segment in a Pascal-like language:

```
if x=y
  then x:=x-10
  else while y>10 do
    if y<x
      then y:=y+1
      else y:=func(x)
```

   Translate the code segment into an abtract syntax tree, quadruples, and postfix code.

10. (3p) **Intermediate Code Generation**

    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 L3
```