Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2019-08-29
Sal (1)	T1(6)
Tid	8-12
Utb. kod	TDDB44
Modul	TEN1
Utb. kodnamn/benämning **Modulnamn/benämning**	Kompilatorkonstruktion Tentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	11
Jour/Kursansvarig Ange vem som besöker salen	Martin Sjölund
Telefon under skrivtiden	+46 13 28 6679
Besöker salen ca klockan	ca 9:30
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Veronica Kindeland Gunnarsson 013-28 56 34 veronica.kindeland.gunnarsson@liu.se
Tillåtna hjälpmedel	Engelsk ordbok Miniräknare
Övrigt	
Antal exemplar i påsen	

Tentamen/Exam

TDDB44 Kompilatorkonstruktion / Compiler Construction

2019-08-29, 08:00 – 12:00

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 *not* ordered according to difficulty.
- The exam is designed for 40 points in 240 minutes. You may thus plan 6 minutes per point.
- Grading: U, 3, 4, 5 resp. Fx, C, B, A (ECTS grades given on request).
- The preliminary threshold for passing (grade 3/C) is 20 points.

1. (3p) **Compiler Structure and Generators**

   (a) (2p) What are the generated compiler phases and what are the corresponding formalisms (mention at least 5 phases and 3 formalisms) when using a compiler generator to generate a compiler?

   (b) (1p) Most modern compilers have not just one, but several intermediate representations (IR). What is the advantage of having more than one IR, and what could be the drawback?

2. (5p) **Top-Down Parsing**

   (a) (4.5p) Given a grammar with nonterminals L, E, F and the following productions:

```
L ::= L a
L ::= E F b
L ::= E F f
E ::= E c
E ::= d
F ::= E e
F ::= ε
```

   where L is the start symbol, a, b, c, d, e and f 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?

3. (3p) **LR parsing**

Use the SLR(1) tables below to show how the string `xyxizxx` 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
2. X ::= x X y
3.     | x Y x
4.     | x Y z
5. Y ::= x X y
6.     | y X i
7.     | i
```

Tables:

	Action					GOTO		
State	$	x	y	z	i	S	X	Y
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	*	*	*	*

4. (3p) **LR parser construction**

   Given the following grammar G for strings over the alphabet {a,b,c,d} with nonterminals S, A and B, where S is the start symbol:

   1. S ::= A
   2. A ::= a A b
   3. | a B a
   4. | a B c
   5. B ::= a A b
   6. | b A d
   7. | 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.

   If it is not: describe where/how the problem occurs.

5. (3p) **Symbol Table Management**

   The C language allows static nesting of scopes for identifiers, determined by blocks enclosed in braces. Given the following C program fragment (some statements are omitted):

```
int m;
int main( void ) {
  int i;
  for (j=0; j<i; j++) {
    int j, m;
    if (j==m) {
      int j;
      i = m * j;
    }
  }
}
```

(a) For the program point containing the assignment `i = m * j`, show how the program variables are stored in the symbol table if the symbol table is to be realized as a hash table with chaining and block scope control. Assume that your hash function yields value 3 for `i`, value 1 for `j` and `m`, and value 4 for `main`.

(b) Show and explain how the right entry of the symbol table will be accessed when looking up identifier `m` in the assignment `i = m * j`.

(c) After code for a block is generated, one needs to get rid of the information for all variables defined in the block. Given a hash table with chaining and block scope control as above, show how to "forget" all variables defined in the current block, without searching through the entire table.

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. (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.

8. (3p) **Memory management**

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

   (b) (1p) What happens on the stack at function call and at function return?

   (c) (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 fib(x)>4711
  then z=0
  else repeat
          y=fac(x);
          x=x+y;
       until x>50000
```

   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):

```
L1: x:=x+1
L2: x:=x+1
L3: x:=x+1
    if x=1 then goto L5
    if x=2 then goto L1
    if x=3 then goto L3
L4: x:=x+1
L5: x:=x+1
    if x=4 then goto L4
```

11. (6p) **Code Generation for RISC, etc.**

(a) (1.5p) 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?

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

(c) (2p) What is branch prediction and when is it used? Why is this important for pipelined processors?

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