UNIVERSITY OF COPENHAGEN
Department of Computer Science
Course Responsible: Cosmin Oancea
Cell: +45 23 82 80 86
cosmin.oancea@diku.dk

Exam for Course "Implementering af programmeringssprog" ("IPS", NDAB16006U), 5th of April 2017

Exam Policy and Aids

- Exam lasts for four hours.
- The exam hand-out consists of 20 pages, from which the last five pages are blank and intended as scrap paper. ***Students are asked to write their answers on the exam hand-out itself.***
- ***Students are asked to write their name, their KU id, their exam number, and total number of hand-in pages on every page of the exam that they are handing in.***
- Students ***are allowed*** to bring and freely use any written material they wish.
- Students **are not allowed** to use any kind of electronic device (e.g., cell-phones, laptop, etc.), or to communicate with each other.
- Students are allowed to use pencil, eraser, pen, etc.

Task Sets in the Exam The exam consists of four sections that sum-up to 100 points:

- 13 *True/False Statements* 13 points in total (1 point each).
  Solved by circling True if you think the statement is true and False otherwise.
- 6 *Multiple-Choice Questions* 12 points in total.
  Solved by circling exactly one answer. If none of the answers are correct, when available, chose "d) none of the above".
- 12 *Short-Answer Questions* 45 points in total, where enough space is provided to answer each question just below it. We advise you to work out the solution on scrap paper and to write a concise answer (the essential) in the provided space.
- 3 *Longer-Answer Questions* 30 points in total (10 points each), where enough space is provided to answer each question just below it. We advise you to work out the solution on scrap paper and to write a concise answer (the essential) in the provided space.

To keep the exam solution tidy, students are encouraged to use pencil (and eraser) for the questions that they are unsure of, and to use pen for the final solution.

Exam for Course "Implementering af programmeringssprog" (IPS)

Course Responsible: Cosmin Oancea, cell: +45 23 82 80 86

Block 3, 2016/17

I. True/False Questions. Total: 13 points, 1 point each question.

1. With a *DFA*, the current state together with the current input character determine uniquely the (transition to the) next state.	True	False
2. In the worst case, *NFA* tests membership in time proportional only with the input-string size (and a small constant).	True	False
3. Arbitrary-nested function calls, e.g., `f( g( h( q(...) ) ) )`, can be expressed with a *regular expression*.	True	False
4. The language $L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m\mathtt{d}^m \mid n,m \in \mathbb{N}\}$, is representable with a *context-free grammar CFG*. (The words in L start with an arbitrary number of `a`s followed by the same number of `b`s, followed by an arbitrary number of `c`s followed by the same number of `d`s.)	True	False
5. In the worst case, NFA-to-DFA translation may quadratically increase the size (number of states) of the *resulted DFA*.	True	False
6. The *call-by-name* evaluation strategy might terminate in some cases in which *call-by-value* would not.	True	False
7. The statement `if a > b then a := b-a else a := a-b` is a valid *three-address code* statement, i.e., a valid statement in the intermediate language used by the book/lecture slides.	True	False
8. *Callee-saves* registers are preferably used to store variables that are dead (in the caller) after the function call.	True	False
9. Fasto's *interpreter* is strongly and statically typed.	True	False
10. Fasto and the book's *code generation* uses a variable symbol table (`vtable`) that binds variable names in the source language to their types.	True	False
11. In statement `M[a] := a + b`, the kill set is `{a}` *and* the gen set is `{a,b}` (referring to liveness analysis).	True	False
12. With an *intermediate language*, translating `m` languages to `n` hardware architectures requires implementing only `m+n` translators (rather than `m*n`).	True	False
13. After executing the *MIPS instruction* `ADDI $2 $0 131072`, the value stored in register `$2` will not be `131072` because MIPS instructions require constants less than 2^{16}.	True	False

II. Multiple-Choice Questions. Total 12 points.

Choose exactly one answer. When available, if no answer is correct choose "none of the above"!

1. (2pts) Consider the *C-like* pseudo-code below:

```
void main() {
 int x = 3;
 f(x, x);
 print(x);
}
```

```
void f(int a, int b) {
    a = a - b;
    b = 2*b + a;
    a = b;
}
```

Under which *calling convention* does `main()` print 6?

a) call by value.

b) call by reference.

c) call by value result.

d) none of the above.

2. (2pts) Consider the *C-like* pseudo-code below:

```
int x = 100;
void g() { print(x); }
void main(int y) {
    f(5, y);
}
```

```
void f(int x, int y) {
    if (y > 4) {            g(); }
    else       { int x = 9; g(); }
}
```

Under *dynamic scoping*, what does `main(3)` print?

a) prints 3.

b) prints 5.

c) prints 9.

d) prints 100.

3. (2pt) Consider the *Fasto* program below:

```
fun [[int]] main() = let n = read(int) in
                     let a = iota(n)   in
                     replicate(n, a)
```

Assuming an integer (and address) value is represented on 1 word (one word = 4 bytes), how many words will be allocated on *the heap* during the execution of `main`?

a) `(n+1)*(n+1)`

b) `n*(n+1)`

c) `2*(n+1)`

d) `2*n`

4. (2pt) Which word does not belong to the regular language $(a\ (b|a)^*\ b) \mid ((ab)^*\ a)$

   a) *aaaab*
   b) *aabba*
   c) *ab*
   d) *ababa*

5. (2pt) Which of the CFG-defined languages below are regular, i.e., expressible with a *regular expression*? (S and B are non terminals, and $a, b, c, d, (,)$ are terminals.)

   a) $S \rightarrow\ abS \mid B$
      $B\ \rightarrow\ cB \mid dB \mid \varepsilon$
   b) $S \rightarrow a(S) \mid b$
   c) $S \rightarrow aSa \mid bSb \mid cSc \mid \varepsilon$
   d) none of the above.

6. (2pt) In the *combined caller-callee saves strategy* for implementing function calls, which of the following is true with respect to saving registers?

   a) *caller-saves* registers ideally hold variables that are live after the function call,
   b) *callee-saves* registers are not saved if the callee does not use them,
   c) function's arguments/parameters are ideally stored in *callee-saves registers*,
   d) none of the above.

III. Short Answers. Total 45 points.

1. (2pts) In the `Lexer.fsl` file of the group project, explain the meaning of the regular expression `['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*` and its corresponding action, i.e., what does function `keyword` do? (Maximum 6 lines)

```
rule Token = parse ...
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*}
    { keyword ( Encoding.UTF8.GetString(lexbuf.Lexeme), getPos lexbuf) }
...
let keyword (s, pos) = match s with
                         | "if" -> Parser.IF pos
                         ...
                         | _    -> Parser.ID (s, pos)
```

2. (3pts) Below is a code snippet from `Parser.fsp` file of the group project:

```
%token <(int*int)> OR AND IF THEN ELSE ...
%left OR
%left AND
%nonassoc NOT
%left LTH
%type <Fasto.UnknownTypes.Exp> Exp
Exp : ... | Exp OR Exp { AbSyn.Or ($1, $3, $2) }
```

   a) Explain rule `Exp` $\rightarrow$ `Exp OR Exp`: (i) how is it disambiguated (what does `%left OR` do?), and (ii) what does the rule produce and from what (what are `$1`,`$2`, `$3`)? (max 8 lines)

   b) Show by inserting paranthesis how `not x < y || y < z && x < z` is parsed. (`NOT`, `AND`, `OR` and `LTH` are the terminals denoting the `not`, `&&`, `||`, and < keywords.)

3. (3pts) a) Write below a DFA for the regular expression $0 \mid (\ 1\ (\ (10) \mid 0)^*\ )$

b) For the alphabet Σ = {0,1,...,9}, write a regular expression describing the language of natural numbers divisible by 5. (For simplicity, numbers starting with 0 are legal.)

4. (2pts) Consider the grammar $S \rightarrow SS+ \mid SS* \mid$ a

a) Can aa+*a be derived from S? YES NO
If it can, show the derivation on the line below:
$S \Rightarrow$

b) Can aa*a+ be derived from S? YES NO
If it can, show the derivation on the line below:
$S \Rightarrow$

5. (3pts) The following code implements the interpretation of the `reduce` operator in Fasto.
Explain the code (lines $4-9$) in maximum 6 lines of text.

```
1.  let rec evalExp (e : UntypedExp, vtab : VarTable, ftab : FunTable) : Value =
2.    match e with ...
3.      | Reduce (farg, ne, arrexp, tp, pos) ->
4.          let arr  = evalExp(arrexp, vtab, ftab)
5.          let nel  = evalExp(ne,     vtab, ftab)
6.          match arr with
7.            | ArrayVal (lst,tp1) ->
8.                 List.fold (fun acc x -> evalFunArg (farg, vtab, ftab, pos, [acc;x])) nel lst
9.            | _ -> raise (MyError("Third argument of reduce is not an array: " + ppVal 0 arr
```

6. (3pts) The code below shows the implementation of function `evalFunArg`, which is used in the evaluation of Fasto's operators such as `reduce` – see previous task.

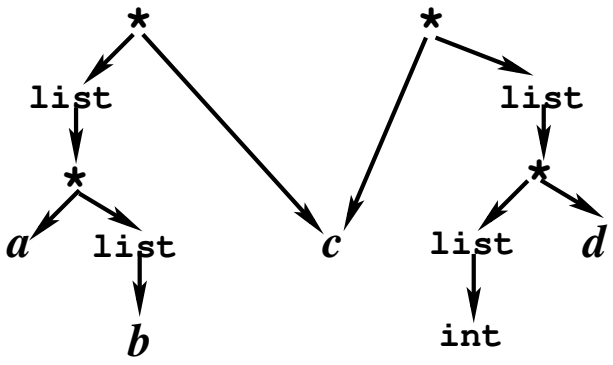
```
10. and evalFunArg  ( funarg: UntypedFunArg, vtab: VarTable, ftab: FunTable
11.                 , callpos: Position, aargs: Value list) : Value =
12.   match funarg with
13.     | (FunName fid) ->
14.         match SymTab.lookup fid ftab with
15.           | None   -> raise (MyError("Call to known function "+fid, callpos))
16.           | Some f -> callFunWithVtable(f, aargs, SymTab.empty(), ftab, callpos)
17.     | Lambda (rettype, parms, body, fpos) ->
18.         callFunWithVtable ( FunDec ("<anonymous>", rettype, parms, body, fpos)
19.                           , aargs, vtab, ftab, callpos )

20. and callFunWithVtable ( fundec: UntypedFunDec, aargs: Value list, vtab  : VarTable
21.                       , ftab  : FunTable, pcall : Position ) : Value =
22.   let (FunDec (fid, rtp, fargs, body, pdcl)) = fundec
23.   let vtab1 = bindParams (fargs, aargs, fid, pdcl, pcall)
24.   let vtab2 = SymTab.combine vtab1 vtab
25.   let res   = evalExp (body, vtab2, ftab)
26.   if typeMatch (rtp, res) then res else raise (MyError("...", pcall))
```

a) Explain why on code line 16 `callFunWithVtable` is called with an empty variable symbol table (Maximum 4 lines).

b) Explain why on code line 18 `callFunWithVtable` is called with the current variable symbol table `vtab`. Perhaps show an example. (Maximum 7 lines).

7. (4pts) Do the types `list(`*a* `* list(`*b*`)) * `*c* and *c* `* list(list(char) * `*d*`)` unify?
If they do what is the unified type, i.e., the most generic unifier (MGU), *and also* what are *a*, *b*, *c*, and *d* after unification?
(The graph-representation of the two types is given below. Assume that *a*, *b*, *c* and *d* are greeks, i.e., universally quantified type variables, and that `list` and `*` are type constructors.)



8. (3pts) This task refers to machine-code generation by the greedy technique that pattern matches the longest available intermediate-language (IL) pattern. Assume that the IL supports the `>=` binary operator, which returns 1 if the first argument is greater-or-equal than the second argument and 0 otherwise. Assume that the IL also supports the `!` unary operator, which is defined only on `0` and `1` resulting in `1` and `0`, respectivelly (i.e., logical not).

a.) Fill in an *efficient* MIPS translation in the blank entries of the table below (avoid jumps):

`x := `r_y `>=` r_z r_q `:= !x`^{last}	
r_x `:=` r_y `>=` r_z	`slt` r_x`,` r_y`,` r_z `xori` r_x`,` r_x`, 1`
r_x `:= !`r_y	
r_d `:=` r_s `+ k`	`addi` r_d`,` r_s`, k`
`IF` r_s `=` r_t `THEN lab`$_t$ `ELSE lab`$_f$ `lab`$_t$`:`	`bne` r_s`,` r_t`, lab`$_f$ `lab`$_t$`:`
`IF` r_s `=` r_t `THEN lab`$_t$ `ELSE lab`$_f$	`beq` r_s`,` r_t`, lab`$_t$ `j lab`$_f$`:`

(MIPS `slt` r_a`,` r_b`,` r_c instruction will set r_a to 1 if r_b is less than r_c and to 0 otherwise. MIPS `xori` instruction performs bitwise exclusive-or between a register and a constant.)

b.) Assuming `vtable = [x→r_x, y→r_y, q→r_q]`, translate the IL code below to MIPS code (with symbolic registers), by *using the previous table*. Write your MIPS code on the right side.

```
y := q >= x
x := !y
IF y=q THEN lab1 ELSE lab2
x := x + 1
```

9. (4pts) Assuming that

(i) the variable symbol table (`vtable`) is: `[x→` r_x`, y→` r_y`]`,

(ii) operator `||` denotes the logical-or operator, and should be translated with jumping (short-circuited) code,

(iii) operators are named the same in the source and IL language,

Translate the code below to three-address code (IL of the book/slides). Try to stay as close as possible to the intermediate-language (IL) translation algorithm in the book/slides.

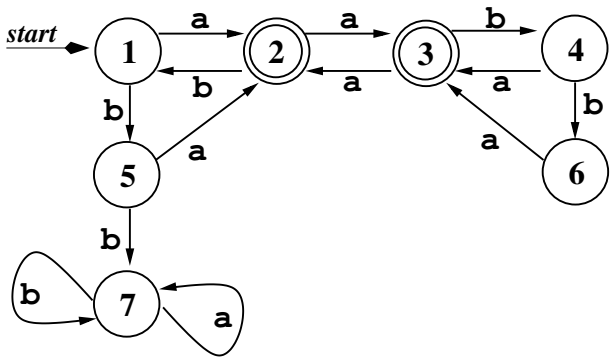
```
repeat {
    y := y - 1;
    x := x + 2;
} until ((y<0) || (x>100))
```

10. (5pts) With the DFA below do the following:

a.) Add a dead state and corresponding transitions to make the `move` function complete.

b.) Minimize the resulting DFA; show all group tables (about seven of them).

c.) Draw the minimized DFA and remove the states known as dead.



11. (2pts) Eliminate the left-recursion of the grammar bellow and write down the resulted equivalent CFG:

$E \rightarrow E\ X\ \texttt{a}$
$E \rightarrow E\ X\ \texttt{b}$
$E \rightarrow \texttt{a}$
$E \rightarrow \texttt{b}$
$X \rightarrow \texttt{a}\ X$
$X \rightarrow \varepsilon$

12. (5pts) Compute `Follow(`Y`)` and the lookahead sets for each production of the grammar:

$S \rightarrow X\ \$$
$X \rightarrow \texttt{a}\ Y$
$Y \rightarrow X\ Z$
$Y \rightarrow \varepsilon$
$Z \rightarrow \texttt{b}\ Y$

`Follow(`Y`) =`

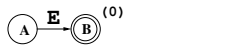
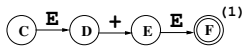
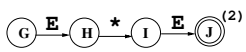
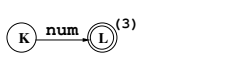
`LookAhead(`$X \rightarrow \texttt{a}\ Y$`) =`

`LookAhead(`$Y \rightarrow X\ Z$`) =`

`LookAhead(`$Y \rightarrow \varepsilon$`) =`

`LookAhead(`$Z \rightarrow \texttt{b}\ Y$`) =`

13. (6pts) Consider the grammar bellow, together with its NFA for SLR parsing (ε connections not shown):

$E' \rightarrow E$	(0)	
$E \rightarrow E + E$	(1)	
$E \rightarrow E * E$	(2)	
$E \rightarrow$ num	(3)	

The SLR parse-table below has as the first column the sets of NFA states obtained by applying the subset-construction algorithm to the original NFA, and the second column contains the corresponding DFA states. The go actions are already filled.
Assuming that `Follow(E) = {'$', '+', '*'}`, do the following:

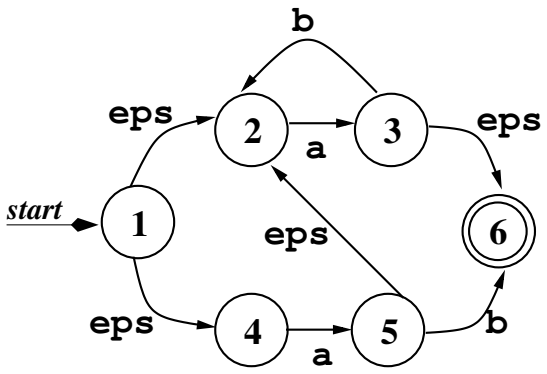
1. Fill the SLR parse table below with all the *shift and reduce actions*.

NFA states	DFA state	+	*	num	$	E
{A,C,G,K}	0					g1
{B,D,H}	1				accept	
{L}	2					
{E,C,G,K}	3					g5
{I,C,G,K}	4					g6
{F,D,H}	5					
{J, D, H}	6					

2. Briefly explain how to resolve the resulted *shift-reduce conflicts*, assuming that $*$ has higher precedence than $+$ and both $*$ and $+$ are left associative. (Show it in the table by crossing the undesired action.)

IV. Longer Answers. Total 30 points.

1. (10pts) Convert the non-deterministic finite automaton (NFA) below into a deterministic finite automaton (DFA) using the *subset-construction* algorithm. Derive the start state and the `move` function for all ($\sim$6) pairs of DFA state and input character. Then draw the resulting DFA on the right-hand side of the original NFA. (In the Figure, `eps` stands for ε transition.)



2. (10pts) This task refers to type checking. Assume we want to extend FASTO with a new second-order array combinator named `fold`, *whose type is* `fold :` $(\ (\beta * \alpha \to \beta)\ *\ \beta\ *[\alpha]\ )\ \to\ \beta$.

Its semantics is: `fold(f, acc, [`a_1`,...,`a_n`) = f( ... f( f(acc,`a_1`), `a_2` ) ... `a_n`)`, i.e., it applies the function argument to `acc` and the first element of the input array, then to the result and the second element, and so on.

Your task is to write in the table below the type-checking pseudocode for `fold`, i.e., the high-level pseudocode for computing the result type of `fold(f, acc_exp, arr_exp)` from the types of `f`, `acc_exp` and `arr_exp`, together with whatever other checks are necessary. Assume that the function parameter of `fold` is passed only as a string denoting the function's name, i.e., do *NOT* consider the case of anonymous functions (lambda expressions). Try to stay close to the notation used in the textbook/lecture slides. Give self-explanatory error messages.

The result of $Check_{Exp}$ is only the type of the `fold(f,acc_exp,arr_exp)` expression.

$Check_{Exp}(Exp, vtable, ftable) =$ `match` *Exp* `with`	

`fold( f`
`, acc_exp`
`, arr_exp`
`)`	

3. (10pts) Liveness-Analysis and Register-Allocation Exercise. Given the following program:

```
    F(x, y) {
1:   LABEL begin
2:   a := x + y
3:   y := y + a
4:   b := y % 3
5:   x := x - b
6:   IF 0 < x THEN end ELSE begin
7:   LABEL end
8:   RETURN y
    }
```

1 Show **succ**, **gen** and **kill** sets for instructions `1-8`.

2 Compute **in** and **out** sets for every instruction;
stop after two iterations.

3 Draw the interference *table* (Stmt | Kill | Interferes With).

4 Draw the interference *graph* for `x`, `y`, `a`, `b`.

5 Color the interference graph with 3 colors
(show the stack as a table: Node | Neighbors | Color).