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), 11th of April 2018

---

**Exam Policy and Aids**

- Exam lasts for four hours.
- The exam hand-out consists of 15 pages. The exam is printed on one side, so if your answers requires more space you may write on the other (blank) side as well. You may also staple more pages to the hand-in. ***Students are asked to write their answers on the exam hand-out itself, i.e., to write the answer in the close neighborhood of where the question was asked.*** (Once filled in, the exam will be handed in for marking, of course.)
- ***Students are asked to write their exam number, and the total number of hand-in pages, preferably on several pages of their exam.***
- Students ***are allowed*** to bring and freely use any written/printed material they wish.
- Students **are not allowed** to communicate with each other (or with anybody else, except for the exam superivors, of course).
- 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".
- 11 *Short-Answer Questions* 45 points in total, where enough space is provided to answer each question just below (or besides) 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, 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, 2018

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

| | | |
|---|---|---|
| 1. Overall, I think this exam was fair! (Answer last; any answer gets 1 point) | True | False |
| 2. With an *NFA*, the current state together with the current input character determines uniquely the (transition to the) next state. | True | False |
| 3. In the worst case, *NFA* tests membership in time proportional with the product between the size of the input string and the size of the NFA. | True | False |
| 4. The language $L = \{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^p \mid n,m,p \in \mathbb{N}\}$ is representable with a regular expression. (Words in $L$ start with an arbitrary number of `a`s, followed by an arbitrary number of `b`s, followed by an arbitrary number of `c`s.) | True | False |
| 5. If $\alpha$ and $\beta$ are regular expressions then the language obtained from *subtracting* the language of $\beta$ from the language of $\alpha$ is also a regular expression (i.e., the words that belong to $\alpha$ but *not* to $\beta$). | True | False |
| 6. Under *call-by-value* calling convention, the value of `a` just after the call to `f(a)` may be different than the value of `a` just before the call. | True | False |
| 7. In Fasto, *interpretation* uses a variable symbol table (`vtab`) that binds variable names to their values. | True | False |
| 8. In Fasto, *type checking* uses a function symbol table (`ftab`) that binds function names to their type signature. | True | False |
| 9. F# is a dynamically typed language. | True | False |
| 10. The statement: `if (a < b) && (c < d) then label1 else label2` is a valid *three-address code* statement, i.e., a valid statement in the intermediate (target) language used by the book/lecture slides. | True | False |
| 11. *Caller-saves* registers are ideally used to store the variables that are live after the function call (in the caller). | True | False |
| 12. In statement `a := a + b`, the kill set is `{a}` *and* the gen set is `{b}` (referring to liveness analysis). | True | False |
| 13. Regular expressions can define languages that cannot be possibly implemented by a DFA. | 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 9?

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(8)` print?

a) prints 3.

b) prints 5.

c) prints 9.

d) prints 100.

3. (2pts) The *interpretation* of the FASTO program below

```
fun [int] main() = let x = reduce(fn int (int x, int y) => x+y, 0, iota(5))
                   in if x < 3 then x else iota(x)
```

results in:

a) an error because the types of the `then` and `else` branches do not match, i.e., `int` vs. `[int]`

b) integer value `6`

c) integer-array value `[0,1,2,3,4,5,6,7,8,9]`

d) integer-array value `[0,1,2,3,4,5]`

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

   a) *bababb*

   b) *aaaa*

   c) *babbb*

   d) *aab*

5. (2pt) The following *language identity*:
   $\{a^n \; b^n \; c^n \mid n \in \mathbb{N}\} \equiv \{a^n \; b^n \; c^m \mid n,m \in \mathbb{N}\} \cap \{a^n \; b^m \; c^m \mid n,m \in \mathbb{N}\}$,
   where $\cap$ denotes language (set) intersection, shows that:

   a) the intersection of two languages expressible by context-free grammars does not necessarily result in a language expressible by a context-free grammar.

   b) the intersection of two languages expressible by regular expressions does not necessarily result in a language expressible by a regular expression.

   c) the intersection of a language expressible by a context-free grammar with a language expressible by a regular expression always results in a language expressible by a regular expression.

   d) the earth has spherical shape ;-)

6. (2pt) In the *combined caller-callee saves strategy* for implementing function calls, which of the following is not true with respect to saving registers?

   a) *caller-saves* registers ideally hold variables that are dead after the function call,

   b) *callee-saves* registers are not saved if the callee does not use them,

   c) function's arguments/parameters are stored in *callee-saves registers*,

   d) variables that are live after the function call are preferably stored in *callee-saves* registers.

**III. Short Answers. Total 45 points.**

1. (3pts) Below is a code snippet from `Parser.fsp` file of the group project:

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

   a) Explain rule `Exp` $\rightarrow$ `Exp AND Exp`: (i) how is it disambiguated (what does `%left AND` 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.)

2. (3pts) The following code implements the interpretation of the `map` operator in Fasto.
   Explain the code (lines 4 – 10) in maximum 6 lines of text.

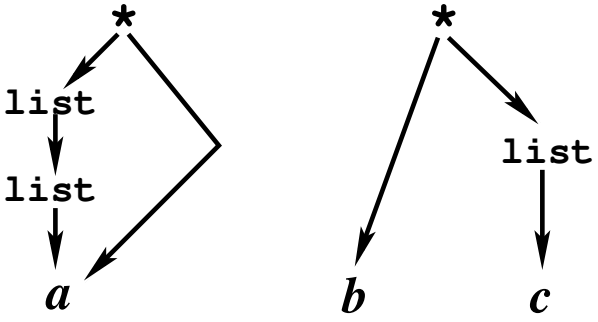
```
1.  let rec evalExp (e : UntypedExp, vtab : VarTable, ftab : FunTable) : Value =
2.    match e with ...
3.      | Map (farg, arrexp, _, _, pos) ->
4.          let arr  = evalExp(arrexp, vtab, ftab)
5.          let farg_ret_type = rtpFunArg farg ftab pos
6.          match arr with
7.            | ArrayVal (lst,tp1) ->
8.                 let mlst = List.map (fun x -> evalFunArg (farg, vtab, ftab, pos, [x])) lst
9.                 ArrayVal (mlst, farg_ret_type)
10.           | _ -> raise (MyError("Second argument of map is not an array", pos))
```

3. (6pts) This task refers to type checking FASTO's `scan` (second-order) array operator.
Please recall that `scan` has type: $(\ (\alpha * \alpha \rightarrow \alpha)\ *\ \alpha\ *[\alpha]\ )\ \rightarrow\ [\alpha]$

Your task is to write in the table below the type-checking pseudocode for `scan`, i.e., the high-level pseudocode for computing the result type of `scan(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 `scan` 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 $Check_{Exp}$ should be the type of the input expression.

| $Check_{Exp}(Exp, vtable, ftable) =$ `case` $Exp$ `of` | |
|---|---|
| `scan(f, acc_exp, arr_exp)` | |

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




5. (3pts) This task refers to machine-code generation by the greedy technique that pattern matches the longest available intermediate-language (IL) pattern. Using the intermediate, three-address language (IL) from the book (slides) and the IL *patterns*:

| | |
|---|---|
| `q := r`$_s$ `+ k`<br>`M[q`$^{last}$`] := r`$_t$ | `sw r`$_t$`, k(r`$_s$`)` |
| `M[r`$_s$`] := r`$_t$ | `sw r`$_t$`, 0(r`$_s$`)` |
| `r`$_d$ `:= r`$_s$ `+ r`$_t$ | `add r`$_d$`, r`$_s$`, r`$_t$ |
| `r`$_d$ `:= r`$_s$ `+ k` | `addi r`$_d$`, r`$_s$`, k` |
| `IF r`$_s$ `= r`$_t$ `THEN lab`$_t$ `ELSE lab`$_f$<br>`lab`$_f$`:` | `beq r`$_s$`, r`$_t$`, lab`$_t$<br>`lab`$_f$`:` |
| `IF r`$_s$ `= r`$_t$ `THEN lab`$_t$ `ELSE lab`$_f$ | `beq r`$_s$`, r`$_t$`, lab`$_t$<br>`j lab`$_f$`:` |
| `lab:` | `lab:` |

and assuming `vtable = [x`$\rightarrow$`r`$_x$`, y`$\rightarrow$`r`$_y$`, q`$\rightarrow$`r`$_q$`]`, translate the IL code below to MIPS code (with symbolic registers). Write your MIPS code on the right-hand side.

```
y := q + 4
M[y^last] := x
IF x = q^last THEN lab1 ELSE lab2
lab2:
```

6. (5pts) Assuming that

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

(ii) operator `&&` denotes the logical-and 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 - x;
    x := x + 5;
} until ((y<x) && (x>100))
```

7. (5pts) **This task refers to the implementation of copy/constant propagation in Fasto (Let and Var cases).**

```
type Propagatee = ConstProp of Value
                | VarProp   of string
type VarTable = SymTab.SymTab<Propagatee>

let rec copyConstPropFoldExp (vtable: VarTable) (e: TypedExp) =
1. match e with ...
2.   | Let (Dec (name, e, decpos), body, pos) ->
3.         let e' = copyConstPropFoldExp vtable e
4.         let vtab' = match e' with   (* bind me to vtable *)
5.                       | Var      (x, _) -> ...

6.                       | Constant (v, _) -> ...

7.         let body' = copyConstPropFoldExp ...

8.         Let( Dec (name, e', decpos), body', pos)

9.   | Var (name, pos) ->
10.        match SymTab.lookup name vtable with
11.          | Some (VarProp   newname) -> ...

12.          | Some (ConstProp newval ) -> ...

13.          | _                       -> Var (name, pos)
```

Your task is to complete the implementation by filling in the dots at lines 5, 6, 7, 11, and 12 and to briefly explain the rationale of your additions. (A small example could be useful.)

8. (3pts) **This task refers to the implementation of constant folding in Fasto (And case).**

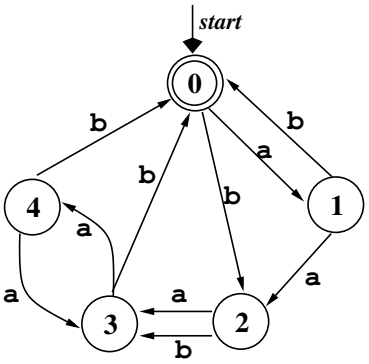
```
let rec copyConstPropFoldExp (vtable: VarTable) (e: TypedExp) =
match e with ...
  | And (e1, e2, pos) =>
1.    let e1' = copyConstPropFoldExp vtable e1
2.    let e2' = copyConstPropFoldExp vtable e2
3.    match (e1', e2') with
4.      | (Constant (BoolVal true, _), _) -> ...

5.      | (_, Constant (BoolVal true, _)) -> ...

6.      | (Constant (BoolVal x, _), Constant (BoolVal y, _)) -> ...

7.      | _ -> And (e1', e2', pos)
```

Your task is to complete the implementation of constant folding for an `And` expression by filling in the dots at lines 4, 5 and 6, and to briefly explain to what algebraic simplifications each case corresponds to (1 line each).

9. (5pts) Minimize the DFA below:

a.) show all group tables and how they get split, and

b.) draw the minimized DFA.



10. (2pts) Eliminate the left-recursion of the grammar below and write down the resulted equivalent CFG:

$E \rightarrow E\ E\ +$

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow$ `num`

11. (6pts) Consider the grammar below together with the associated operators precedence and associativity:

| | | |
|---|---|---|
| $E \rightarrow E$ `pow` $E$ | (1) | `pow` binds tighter than <, |
| $E \rightarrow E$ `\|\|` $E$ | (2) | < binds tighter than `\|\|` |
| $E \rightarrow E < E$ | (3) | `pow` is right associative, |
| $E \rightarrow$ `id` | (4) | < is non associative, and |
| | | `\|\|` is left associative. |

Do the following:

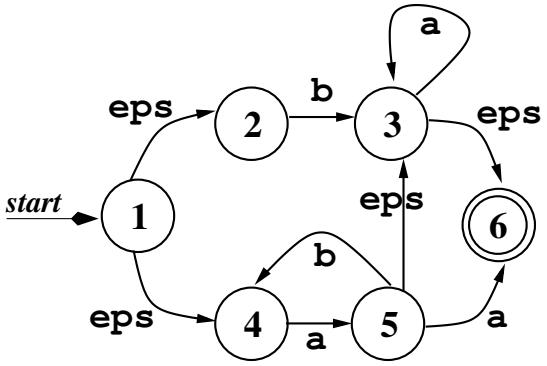
1 **Rewrite the Grammar to be Unambiguous**: (Note that since < is declared non-associative, the resulting grammar will accept a slightly modified language; that's OK!)

2 Assuming the SLR parse-table (excerpt) below of the **original** grammar, **resolve the shift-reduce ambiguity** generated by the operators `pow` and `||` according to the declared precedence and associativity, i.e., keep one shift or one reduce per table entry. Briefly explain your choice: 1-2 lines for each. (Remember that `r`$n$ in the table denotes a reduce action with grammar rule number `(n)` and `sk`$_j$ denotes a shift action to some state `k`$_j$).

| | pow | ... | \|\| |
|---|---|---|---|
| state i | sk$_1$ r1 | ... | sk$_2$ r2 |
| ... | ... | ... | ... |
| state j | sk$_3$ r2 | ... | sk$_4$ r1 |

**IV. Longer Answers. Total 30 points.**

1. (9pts) 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$8) pairs of DFA state and input character. State which DFA states are final (accepting), then draw the resulting DFA on the right-hand side of the original NFA. (In the Figure, `eps` stands for $\varepsilon$ transition.)



**2. (9pts) For the grammar below, fill in the blanks by computing `Follow(`$Y$`)`, the lookahead sets for each rule, and by writing the pseudocode for the corresponding recursive-descent parser.**
$S$, $X$ and $Y$ are non-terminals and `num`, `id`, `$`, `[`, `]`, `+` and `*` are terminals (of course).

$S \rightarrow X$ `$`
$X \rightarrow$ `[` $X$ `]` $Y$
$X \rightarrow$ `num` $Y$
$Y \rightarrow$ `+ id` $Y$
$Y \rightarrow$ `*` `num` $Y$
$Y \rightarrow \varepsilon$

`Follow(`$Y$`) =`

`LookAhead(`$S \rightarrow X$ `$`$\varepsilon$`) =`

`LookAhead(`$X \rightarrow$ `[` $X$ `]` $Y$`) =`

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

`LookAhead(`$Y \rightarrow$ `+ id` $Y$`) =`

`LookAhead(`$Y \rightarrow$ `*` `num` $Y$`) =`

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

```
function parseS() =
```

```
function parseX() =
```

```
function parseY() =
```

**3. (12pts) Liveness-Analysis and Register-Allocation Exercise.** Given the following program:

```
   F(a, b) {
1:   LABEL begin
2:   IF a < 3 THEN lab1 ELSE lab2
3:   LABEL lab1
4:   x := a + b
5:   b := a + x
6:   LABEL lab2
7:   y := b - a
8:   a := a / y
9:   z := a - b
10:  IF z < 0 THEN end ELSE begin
11:  LABEL end
12:  RETURN a
    }
```

1 Compute the **in** and **out** sets for every instruction `1-12`; *stop after two iterations*. (If you wish/need, you may also compute/show the **succ**, **gen** and **kill** sets.)

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

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

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