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), 19th of June 2019

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".
- 13 *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 4, 2019

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. There exists a language that can be represented by some nondeterministic finite automaton, but *not* by any context-free grammar.	True	False
3. The language $L = \{\texttt{a}^n\texttt{c}^m\texttt{b}^n \mid n,m \in \mathbb{N}\}$ is representable with a regular expression. (The words in L start with an arbitrary number of as and end with the same number of bs, and in the middle there are an arbitrary number of cs.)	True	False
4. If a language L is represented by an NFA with n states, then any DFA reprezeting L can have at most 2^{n-1} states.	True	False
5. If `x` is a *local variable*, then the statement `x := f(x)+1` will produce the same result under the *call-by-value* and *call-by-reference* calling conventions.	True	False
6. In Fasto, *interpretation* uses a function symbol table (`ftab`) that binds function names to their type signature.	True	False
7. In Fasto and book, *type checking* uses a variable symbol table (`vtab`) that binds variable names to their types.	True	False
8. *Machine code*, for example MIPS, is strongly typed.	True	False
9. The statement: `M[i] = x`, where `i` and `x` are variable names, 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
10. In the *mixed caller-callee saves* strategy, parameters are ideally passed in callee-saves registers.	True	False
11. If statement `i` is `if a<b then` $\texttt{l}_t$ `else` $\texttt{l}_f$ then the immediate successor of statement `i` is `{i+1}`, i.e., `succ[i] = {i+1}` (referring to liveness analysis).	True	False
12. If a local variable `a`, i.e., which is *not a formal parameter*, is found live at the entry of a function, then, as far as the compiler can tell, there exists an execution path on which `a` may be used before being initialized.	True	False
13. The two consecutive intermediate-language instructions: `r2 := r1 + 256;` `r3 := M[r2`^{last}`]` can be always translated to the following MIPS instruction `lw r3, 256(r1)`.	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 - 2*b;
    b = b + 3;

}
```

Under which *calling convention* does `main()` print 0?

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 = 200;
void h() { print(x); }
void main(int y) {
    f(5, y);
}

void f(int x, int y) {
    if (y > 6) {           g(y-x); }
    else       { int x = 9; g(y+x); }
}
void g(int x)  { h(); }
```

Under *dynamic scoping*, what does `main(4)` print?

a) prints `-1`.

b) prints `9`.

c) prints `13`.

d) prints `200`.

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

```
fun int main() = let xs = filter(fn int (int x) => x > 2, {0, 2, 3, 5})
                 in let x = xs[0] in if x < 4 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 `2`

c) integer value `3`

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

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

   a) *bababa*
   b) *baba*
   c) *babbbabb*
   d) *bababbb*

5. (2pts) Which of the CFG-defined languages below are not regular, i.e., cannot be expressed with a *regular expression*?
   (S and B are non terminals, and a, b, c, d are terminals; S is the start symbol.)

   a) $S \rightarrow abS \mid B$
      $B \rightarrow aB \mid cB \mid dB \mid \varepsilon$
   b) $S \rightarrow Sa \mid b$
   c) $S \rightarrow aSa \mid bSb \mid \varepsilon$
   d) none of the above (i.e., they are all regular).

6. (2pts) 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) the *callee-saves* registers that the callee does not actually use are not saved,
   c) function's arguments/parameters are stored in *callee-saves registers*,
   d) none of the above.

III. Short Answers. Total 45 points.

1. (3pts) Draw the NFA corresponding to the following regular expression: $(b\,(a \mid \varepsilon))^* \mid b$.
(Follow the systematic construction in the book as closely as possible, without any optimizations. Don't forget to indicate the initial and final states!)

2. (3pts) Write a regular expression for the language of finite bit strings (characters 0 and 1) that contain no two consecutive 1s. (Examples: ε, 1, 001, 01001; non-examples: 111, 01100.)

   For partial credit (in case your RE is wrong or missing), at least argue convincingly that this language must be regular.

3. (3pts) Give a context-free grammar equivalent to the regular expression $(b\,(a \mid \varepsilon))^* \mid b$ (i.e., the same as in question 1 of this section). Use only pure CFG productions (no '|'s on the right-hand sides). Remember to identify the start symbol!

4. (3pts) Give a context-free grammar for the language $\{\mathtt{c}^n\mathtt{f}^m\mathtt{g}^{n+m} \mid n,m \in \mathbb{N}\}$, which includes, e.g., the words `ccfffggggg` or `ffgg`.

5. (3pts) Consider the following grammar (with start symbol E):

$$\begin{array}{lcl} E & \rightarrow & T \mid E + T \mid var = E \quad \text{(productions 1–3)} \\ T & \rightarrow & var \mid var \texttt{++} \mid \texttt{(}\ E\ \texttt{)} \quad \text{(productions 4–6)} \end{array}$$

(Here, the ++ is considered a single lexical token.) Is this grammar ambiguous? Argue why not, or show that at least one token sequence can be parsed as two different syntax trees.

6. (3pts) Suppose we want to parse arithmetic expressions given by the following grammar (with start symbol Exp_1):

$$\begin{array}{lcl} Exp_1 & \rightarrow & Exp_2 \mid Exp_1 + Exp_2 \mid Exp_1 - Exp_2 \mid - Exp_2 \\ Exp_2 & \rightarrow & Exp_3 \mid Exp_2 * Exp_3 \\ Exp_3 & \rightarrow & num \mid \texttt{(}\ Exp_1\ \texttt{)} \end{array}$$

Here, we have explicitly disambiguated the grammar by using three separate nonterminals Exp_i to express the desired operator precedences and associativities of addition, subtraction, negation, and multiplication.

Would the following be a correct realization of the above grammar in a Fasto-like parser implementation?

```
%left PLUS MINUS
%left TIMES
...
Exp : Exp PLUS Exp  { Plus (($1, $3, $2) }
    | Exp MINUS Exp { Minus ($1, $3, $2) }
    | Exp TIMES Exp { Times ($1, $3, $2) }
    | MINUS Exp     { Negate ($2, $1) }
    | NUM           { Constant (...) }
    | LPAR Exp RPAR { $2 }
;
```

(where the terminals of the grammar (+, -, ...) correspond to the lexer tokens (`PLUS`, `MINUS`, ...) in the obvious way.) Explain why or why not (max 5 lines).

7. (3pts) Find and explain three errors in the *interpreter* implementation of the `scan` operator below. Also, briefly explain how each error should be fixed.
(We recall that the FASTO's `scan` starts the result array with the first element, while F#'s `List.scan` starts from the accumulator element; hence taking the `List.tail` makes sense in the last line, but maybe something else is missing ...)

```
let rec evalExp (e : UntypedExp, vtab : VarTable, ftab : FunTable) : Value =
  match e with
  | ...
  | Scan ( farg, acc_exp, arr_exp, _, pos) ->
      let farg_ret_type = rtpFunArg farg ftab pos
      let acc_val = evalExp (acc_exp, vtab, ftab)
      let arr_val = evalExp (arr_exp, vtab, ftab)
      let res_val = List.scan (fun acc x -> evalFunArg (farg, vtab, ftab, pos, [x])) acc_val arr_val
      List.tail res_val
```

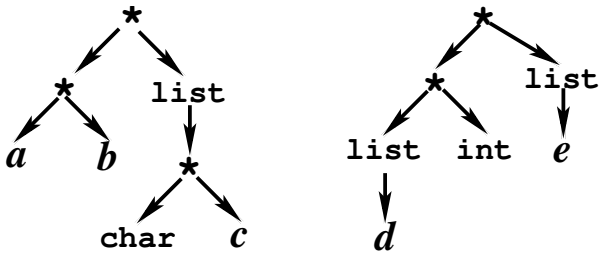
8. (7pts) Assume that we want to extend FASTO with a new operator, named `foldRows`, that semantically applies a (F#-like) fold operation on each row of a two-dimensional array. For example:
`foldRows (⊙, e, { {a1,1, a1,2}, {a2,1, a2,2} })` $\equiv$ `{(e⊙a1,1)⊙a1,2, (e⊙a2,1)⊙a2,2}`
and *most importantly, the type of* `foldRows` is: $((\beta * \alpha \rightarrow \beta) * \beta * [[\alpha]]) \rightarrow [\beta]$.

Your task is to write in the table below (on the next page) the type-checking pseudocode for `foldRows`, i.e., the high-level pseudocode for computing the result type of `foldRows(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 `foldRows` 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 result of the `foldRows` call.

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	

`foldRows(`
`f,`
`acc_exp,`
`arr_exp)`	

9. (4pts) Do the types: `(a*b) * list(char*c)` and `(list(d) * int) * list(e)` unify?
Show the application of the unification algorithm. If they unify, please answer what is the unified type, i.e., the most generic unifier (MGU), *and also* what are *a*, *b*, and *e* after unification?
(The graph-representation of the two types is given below. Assume that *a*, *b*, *c*, *d* and *e* are greeks, i.e., universally quantified type variables, and that `list` and `*` are type constructors.)



10. (2pts) 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` `r`$_t$ `:= M[q`^{last}`]`	`lw r`$_t$`, k(r`$_s$`)`
`r`$_t$ `:= M[r`$_s$`]`	`lw 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`$_f$ `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`$_f$`:`
`lab:`	`lab:`

translate the IL code below to MIPS code. Write your MIPS code on the right-hand side.
(You may use directly `x`, `y`, `q` as symbolic registers, or alternatively, use `r`$_x$, `r`$_y$, `r`$_q$, respectively.)

```
y := q^last + 256
x := M[y]
IF y = x^last THEN lab1 ELSE lab2
lab1:
```

11. (4pts) Assuming that

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

(ii) operator `||` is the logical-or operator, and has to be translated with jumping (short-circuit) code,

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

(iv) parentheses are only used to show how subexpressions are grouped in the AbSyn,

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.

```
if ((y < x) || (x < 100)) {
    y := (x - y) * 4
} else {
    x := (x + 5) + y
}
```

12. (3pts) In the mixed caller/callee-saves strategy with register-passed parameters, as described in the book, the return-address register is classified as callee-save. Justify this choice by giving one or more reasons for why callee-saves would be expected to work better than caller-saves for this purpose. Be as specific as possible. (max 5 lines)

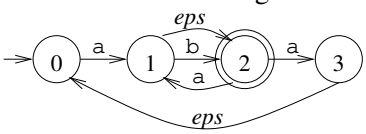
13. (4pts) **This task refers to the implementation of dead-binding removal in Fasto (`Var` and `Let` cases).**

```
1. type DBRtab = SymTab.SymTab<unit>
2.
3. let isUsed (name : string) (stab : DBRtab) : Bool =
4.     match SymTab.lookup name stab with
5.       | None   -> false
6.       | Some _ -> true
7.
8. let recordUse (name : string) (stab : DBRtab) : DBRtab =
9.     match SymTab.lookup name stab with
10.      | None   -> SymTab.bind name () stab
11.      | Some _ -> stab
12.
13.
14. let rec removeDeadBindingsInExp (e : TypedExp) : (bool * DBRtab * TypedExp) =
15.     match e with
16.       | ...
17.       | Var (name, pos) ->
18.
19.
20.       | Let (Dec (name, e, decpos), body, pos) ->
21.             let (eio,    euses,    e'   ) = removeDeadBindingsInExp e
22.             let (bodyio, bodyuses, body') = removeDeadBindingsInExp body
23.
24.
25.
26.
27.
28.
```

Your task is to complete the implementation by filling in the dots at lines $17-19$ for the `Var` case and at lines $23-28$ for the `Let` case, and to briefly explain the rationale of your additions.
It could be useful to give and to explain based on a small example.
(You may use `SymTab.combine` for unioning two symbol tables.)

IV. Longer Answers. Total 30 points.

1. (10pts) Consider the following non-deterministic finite automaton (NFA):



(The NFA alphabet is $\{\mathtt{a}, \mathtt{b}\}$; in the figure, each *eps* stands for an ε-transition.)

a) Convert the NFA into an equivalent deterministic finite automaton (DFA) using the *subset-construction* algorithm. Derive the start state and the `move` function for all pairs of DFA state and input character. Then draw the resulting DFA on the right-hand side of the original NFA. Remember to indicate the accepting state(s). *Hint: if you didn't make any mistakes, your DFA should also contain 4 states.*

b) Minimize the DFA from part (a). Make it clear how the state groups are iteratively refined by the algorithm. Then draw the resulting, minimal DFA.

2. (10pts) Consider the following context-free grammar (with E as the start symbol):

$$\begin{array}{lcl} E & \rightarrow & T \mid E + T \mid E - T \mid - T \\ T & \rightarrow & A \mid T\ ! \\ A & \rightarrow & num \mid (\ E\ ) \end{array}$$

a) Eliminate left-recursion and/or left-factorize the grammar as needed to obtain a CFG for the same language, but suitable for LL(1) parsing. *Hint: if you didn't make any mistakes, the resulting grammar should contain 5 nonterminals and a total of 10 productions.*

b) Add a new start symbol and corresponding production (making the end-of-input symbol $ explicit). Compute *Nullable*, *FIRST*, and *FOLLOW* for all nonterminals in your transformed grammar. You don't have to reproduce the detailed calculations in your exam handin, but it should still be reasonably evident how you obtained the answers. *Hint: when computing Nullable and FIRST, start from the* bottom *of the grammar, and work your way backwards.*

c) For each production, compute its lookahead set, i.e., the set of input symbols that indicate that this production should be chosen by a table-driven or recursive-descent LL(1) parser. Verify that the lookahead sets are non-overlapping whenever there are two or more productions for a nonterminal.

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

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

1 Show **succ**, **gen** and **kill** sets for instructions `1-11`.

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`, `z`, `a`, `b`, `c`.

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