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

Exam for Course "Introduction to Compilers" ("Oversættere", NDAA04011U, B2-2E15), 27th of January 2016

Exam Policy and Aids

- Exam lasts for four hours.
- The exam hand-out consists of 20 pages, from which the last six 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 all answer are correct, when available, chose "d) all 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 "Introduction to Compiler" ("Oversættere")

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

Block 2, Winter 2015/16

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

1.	The language $\{\texttt{a}^n\texttt{b}^n \mid n \in \mathbb{N}\}$, that is, the set of character sequences that start with a number of `a`s, followed by the same number of `b`s is representable with a regular expression (RE).	True	False
2.	The language of all natural numbers is representable with a context-free grammar (CFG).	True	False
3.	If α is a regular expression over alphabet Σ, then its complement, $\Sigma^* - \alpha$, is a regular expression.	True	False
4.	Assuming function `f` does not call any other function, then the call `f(a+b, c+d)` will always give the same result under call-by-value and call-by-reference calling conventions.	True	False
5.	SML is a dynamically typed language.	True	False
6.	A strongly, statically typed language must report at compile time all instances where array indices will be out of bounds.	True	False
7.	Nested comments of arbitrary depth, e.g., `/* ... /* ... */ ... */`, cannot be matched with a regular expression.	True	False
8.	In Fasto, interpretation uses a variable symbol table (`vtab`) that binds variable names to their types.	True	False
9.	`a := b + c * d;` is a valid three-address code statement, i.e., a valid statement in the intermediate language (IL) used by the book/lecture slides.	True	False
10.	In statement `a := a + b` the *kill* set is `{a}` and the *gen* set is `{b}` (referring to liveness analysis).	True	False
11.	An DFA tests membership in time proportional with its number of states and with the input-string size.	True	False
12.	When translating an n-state NFA to a DFA, the size of the DFA may reach the order of 2^n.	True	False
13.	In various passes of Fasto, the variable symbol table (`vtab`) is filled in during the analysis of *let* expressions and formal parameters.	True	False

II. Multiple-Choice Questions. Total 12 points.

Choose exactly one answer. If all answers are correct, when available, choose "all of the above"!

1. (2pt) Which word belongs to regular language $x^+ \ (x|y)^* \ y^+$

   a) *yxxyxy*

   b) *xyyx*

   c) *xxxxy*

   d) *yxxxx*

2. (2pts) If a local variable, i.e., *not* a formal parameter, is *live* at the entry point of a function, then it means that

   a) the variable is only written inside the function.

   b) the variable may be read before being written on at least one of the possible execution paths of the function.

   c) the variable is read before being written on all possible execution paths of the function.

   d) the variable is only read inside the function.

3. (2pt) Which of the following languages, defined via CFGs, are regular, i.e., can be matched by a regular expression?

   a) $S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$

   b) $S \rightarrow aSb \mid \varepsilon$

   c) $S \rightarrow aS \mid B \mid \varepsilon$
   $B \rightarrow bB \mid \varepsilon$

   d) all of the above.

4. (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 not live after the function call,

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

   c) parameters are ideally stored in caller-saves registers,

   d) all of the above.

5. (2pts) Under *dynamic scoping*, what does `main(3)` and `main(6)` print? (See C-like pseudocode 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 = 7; g(); }
}
```

   a) main(3) prints 5 and main(6) prints 7.

   b) main(3) prints 5 and main(6) prints 100.

   c) main(3) prints 100 and main(6) prints 7.

   d) main(3) prints 7 and main(6) prints 3.

6. (2pts) The *interpretation* of the FASTO program below results in:

```
fun int main() = let x = reduce(fn int (int x, int y) =>x+y, 0, iota(4))
                 in if 3 < x then x else iota(4)
```

a) *a lexical error* because `=>x` is neither an **id** nor a keyword (needs a space between `=>` and `x`)

b) *a compile-time error* because the types of the `then` and `else` expressions differ (`int` vs `[int]`)

c) *value* `6` because the `reduce` evaluates to `6` and matches the return type of `main`.

d) *a runtime error* because the result of the `else` branch, which is the one taken, does not match the return type of main.

III. Short Answers. Total 45 points.

1. (2pts) In the `Lexer.lex` file of the group project, explain the meaning of the regular expression `"//" [^ '\n']*` and its action `{ Token lexbuf }`. (Maximum 4 lines)

```
rule Token = parse
        | "//" [^ '\n']*     { Token lexbuf }
       | ...
```

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

```
%token <(int*int)> OR AND IF THEN ELSE ...
%nonassoc letprec
%left OR
%left AND
...
%type <Fasto.UnknownTypes.Exp> Exp
Exp : ...
    | Exp OR Exp { Or ($1, $3, $2) }
    | LET ID EQ Exp IN Exp %prec letprec ...
```

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

   b) Explain what does the use of `letprec` solve? (max 5 lines)

3. (3pts) For the alphabet Σ = {0,1,...,9}, write a regular expression (or a NFA) for the language that contains the natural numbers that *either* (i) are multiples of 10, *or* (ii) in which digit 3 appears exactly twice.

4. (2pts) Consider the grammar

$S \rightarrow \mathtt{a}S\mathtt{b} \mid B$

$B \rightarrow \mathtt{b}B \mid b$

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

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

5. (6pts) The following pseudocode implements the interpretation of a *function call* (FASTO-like)

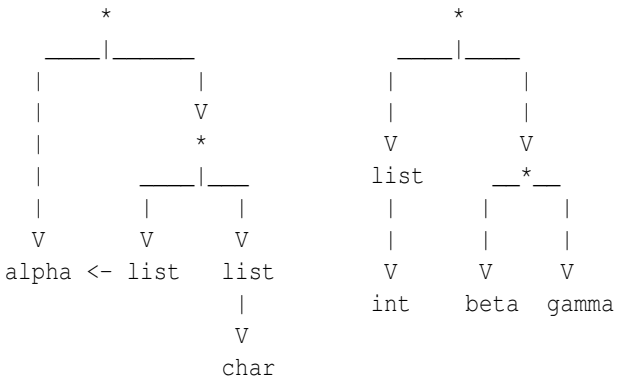
```
1.  fun evalExp ( Apply(fid, args, pos), vtab, ftab ) =
2.     let val evargs = map (fn e => evalExp(e, vtab, ftab)) args
3.     in case SymTab.lookup fid ftab of
4.            SOME f => callFunWithVtable(f, evargs, vtab, ftab)
5.          | NONE   => raise Error(..)
6.     end
7.   | ...
8. and callFunWithVtable ( FunDec (fid, rtp, fargs, body, pdcl), aargs, vtab, ftab) =
9.     let val vtab' = bindParams (fargs, aargs)
10.        val vtab''= SymTab.combine(vtab', vtab)
11.        val res   = evalExp (body, vtab'', ftab)
12.    in  if typeMatch (rtp, res) then res else raise Error(...)
13.    end
```

a) Explain in maximum 4 lines what lines 2, 3 and 4 do.

b) Explain in maximum 4 lines what lines 9, 10, 11 and 12 do.

c) The shown implementation uses STATIC or DYNAMIC scoping? Circle one. Explain in max 2 lines the modifications needed to implement the other form of scoping.

6. (4pts) Do the types `(` α `* (list(`α`) * list(char)) )` and `( list(int) * (`β `*` γ`) )` unify? If they do what is the unified type, i.e., the most generic unifier (MGU), and also who are α, β and γ after unification? The graph-representation of the two types is given below where `alpha`, `beta` and `gamma` stand for α, β and γ, respectively.



7. (3pts) Using the intermediate, three-address language (IL) from the book (slides) and the IL *patterns*:

`q := r`$_s$ `+ k` `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`$_f$ `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`$_f$`:`

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

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

8. (4pts) Assuming that

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

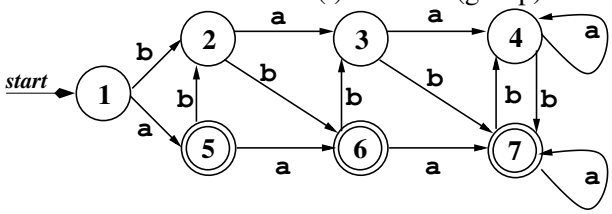
(ii) operator `&&` denotes the logical-and operator, and should be translated with jumping (short-circuited) code,

(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.

```
while(y>0 && x<100) {
    y := y - 1;
    x := x + 2;
}
```

9. (5pts) Minimize the DFA below. (i) show all (group) tables, and (ii) construct the minimized DFA.



10. (2pts) Eliminate the left-recursion of the grammar bellow (and write down the resulted equivalent grammar):
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow$ `num`

11. (5pts) Compute `Follow(`Y`)` and the lookahead sets of the grammar rules deriving Y for the grammar:
$S \rightarrow X$ \$
$X \rightarrow (X) Y$
$X \rightarrow$ `id` Y
$Y \rightarrow$ `num` $+ Y$
$Y \rightarrow$ `id` $* Y$
$Y \rightarrow \varepsilon$

`Follow(`Y`) =`

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

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

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

12. (6pts) Consider the grammar bellow:

$E \rightarrow E \otimes E$ `(1)`
$E \rightarrow E \oplus E$ `(2)`
$E \rightarrow$ `num` `(3)`, where

- $\otimes$ binds tighter than $\oplus$,
- $\otimes$ is left associative,
- $\oplus$ is right associative,

Do the following:

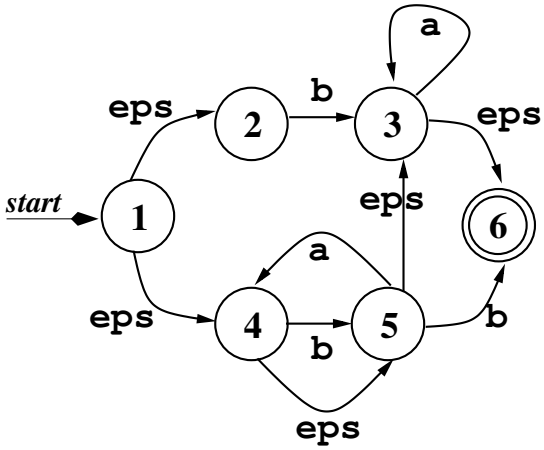
- **Rewrite the Grammar to be Unambiguous**:

- Assuming the SLR parse-table (excerpt) bellow of the **original** grammar, resolve the shift-reduce ambiguity according to the declared operator precedence and asssociativity, 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)`).

	$\oplus$	...	$\otimes$
`state i`	$\texttt{sk}_1$ `r1`	...	$\texttt{sk}_2$ `r2`
...	...	...	...
`state j`	$\texttt{sk}_3$ `r2`	...	$\texttt{sk}_4$ `r1`

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 `move` function for all (∼8) 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.)



2. (10pts) This task refers to type checking FASTO's `reduce` (second-order) operator.

(i) Write the type of `reduce` in FASTO.

(ii) Write in the table below the type-checking pseudocode for `reduce`, i.e., the high-level pseudocode for computing the result type of `reduce(f, n_exp, arr_exp)` from the types of `f`, `n_exp` and `arr_exp`, together with whatever other checks are necessary. Assume that the function parameter of `reduce` 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 `reduce` expression.

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
`reduce(f, n_exp, arr_exp)`	

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

```
   F(a, b) {
 1:   LABEL begin
 2:   IF a < 0 THEN end ELSE continue
 3:   LABEL continue
 4:   t := a + b
 5:   b := b + t
 6:   v := a % 3
 7:   a := a - v
 8:   GOTO begin
 9:   LABEL end
10:   RETURN b
   }
```

1 Show **succ**, **gen** and **kill** sets for instructions `1-10`.

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

3 Draw the interference graph for `a`, `b`, `t`, `v`.

4 Color the interference graph with 3 colors.

5 Show the stack, i.e., the three-column table: Node | Neighbors | Color.