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

Re-Exam for Course "Introduction to Compilers" ("Oversættere", NDAA04011U, B2-2E15), 20th of April 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 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.

Re-Exam for Course “Introduction to Compiler” (“Oversættere”)

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

20th of April 2016

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

1. The language of all palindromes over some finite alphabet is representable with a context-free grammar (CFG). (A palindrome is a word which reads the same backward and forward, e.g., “racecar”.) True False
2. The language $\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^p \mid n,m,p \in \mathbb{N}\}$, that is, the set of character sequences that start with an arbitrary number of `a`’s, followed by an arbitrary number of `b`’s, followed by an arbitrary number of `c`’s, is representable with a regular expression (RE). True False
3. If α and β are regular expression then the language obtained from *subtracting* the language of β from the language of α is also a regular expression (i.e., the words that belong to α but *not* to β). True False
4. Under call-by-value calling convention, the value of `a` after the call to `f(a)` may be different than the value of `a` before the call. True False
5. C is a dynamically typed language. True False
6. A strongly, statically typed (functional) language must report at compile time all instances where the types of the `then` and `else` expressions of an `if` expression are not the same. True False
7. The language that consists of an unbounded, arbitrary number of nested matching open/close parentheses, i.e. {`()`, `(())`, $\cdots$}, can be matched with a context-free grammar. True False
8. In Fasto, interpretation uses a variable symbol table (`vtab`) that binds variable names to their values. True False
9. `if a>0 then x:=x+1 else y:=y+1` 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 `{a,b}` (referring to liveness analysis). True False
11. With a deterministic-finite automaton (DFA), the current state and input character determine *uniquely* the (transition to the) next state. True False
12. One of the benefits of using an intermediate language, is that translating m source languages to n different machine architectures requires implementing only $m \times n$ translators. True False
13. Callee-saves registers are preferably used to store the variables that are not live after a function call. True False

II. Multiple-Choice 6 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 $x^* \ (xy)^* \ y^*$
   a) *xxy*
   b) *yy*
   c) *xxyxyyyy*
   d) all of the above

2. (2pts) Assume liveness analysis is performed at function level (intra-procedural). 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 may be read before being written inside the function.
   b) the variable is only written inside the function.
   c) the variable is only read inside the function.
   d) the variable is not used at all inside the function.

3. (2pt) Which of the following languages, defined via CFGs over alphabet $\sigma = \{a, b\}$, are regular, i.e., can be matched by a regular expression?
   a) $S \rightarrow aSa \mid bSb \mid \varepsilon$
   b) $S \rightarrow Ab$
      $A \rightarrow aA \mid \varepsilon$
   c) $S \rightarrow aSba \mid \varepsilon$
   d) all of the above.

4. (2pt) In the combined caller-callee saves strategy for implementing function calls, which of the following is *not true*?
   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) function parameters are ideally stored in callee-saves registers,
   d) the return address is ideally stored in a callee-saves register.

5. (2pts) Under *dynamic scoping*, what does `main(2)` and `main(8)` print? (See C-like pseudocode below.)

```
int y = 88;                       void f(int y, int x) {
void g() { print(y); }                if (x < 4) {            g(); }
void main(int x) {                    else       { int y = 3; g(); }
    f(9, x);                      }
}
```

   a) main(2) prints 9 and main(8) prints 88.
   b) main(2) prints 9 and main(8) prints 3.
   c) main(2) prints 88 and main(8) prints 3.
   d) main(2) prints 3 and main(8) prints 9.

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

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

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]`

III. Short Answers. Total 45 points.

1. (2pts) In the `Lexer.lex` file of the group project, explain in maximum 4 lines the words recognized by the regular expression `['0'-'9']+` and its action (see code below):

```
rule Token = parse ...
       | ['0'-'9']+     { case Int.fromString (getLexeme lexbuf) of
                               SOME i => Parser.NUM (i, getPos lexbuf) ...}
```

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

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

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) Show by inserting paranthesis how expression `not x < y || y < z && x < z` is parsed? (`NOT`, `AND`, `OR` and `LTH` are terminals denoting the `not`, `&&`, `||`, and < keywords. Max 2 lines.)

3. (3pts) For the alphabet Σ = {a,b}, write a regular expression (or a NFA) describing the language that contains words in which either a appears exactly three times OR the length of the word is a multiple of three.

4. (2pts) Consider the grammar
$S \rightarrow \mathtt{a}S\mathtt{bb} \mid B$
$B \rightarrow \mathtt{b}B \mid b$

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

   b) Can *aabbbbbb* 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. (3pts) Consider the SML code below:

```
let val a = [] in
let val z = if null a
            then ([],        [[1,2],[3]], []      )
            else (["c","a"], a,           [1,2,3])
in  z end;
```

a) Write the type of the `then` expression of the `if` branch (before unification with the `else` expression)

a) Write the type of the `else` expression of the `if` branch (before unification with the `then` expression)

a) Write the type of `z`, i.e., the most generic unifier of the `then` and `else` expressions.

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`:`
`lab:`	`lab:`

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

8. (5pts) Assuming that

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

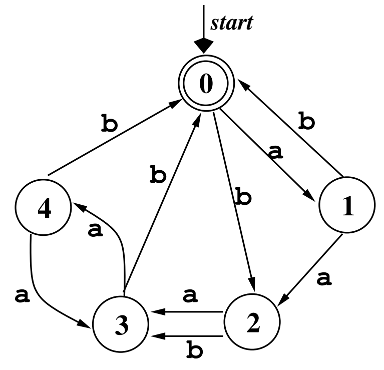
(ii) operator `&&` denotes the logical-and operator, and should be translated with jumping (short-circuited) code,

(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.

```
while( (y > 0) && (x/y < 50) ) {
    y := y - 1;
    x := x + 1;
}
```

9. (5pts) Minimize the DFA below. (i) show all (group) tables and where a group table gets split, and, (ii) after minimization, depict the minimized DFA.



10. (3pts) Eliminate the left-recursion of the grammar bellow (and write the resulted equivalent grammar):

$E \rightarrow E\,E\,+$

$E \rightarrow E\,E\,*$

$E \rightarrow (\,E\,)$

$E \rightarrow \texttt{num}$

11. (4pts) Compute `Follow(`Y`)` and the lookahead sets of the grammar rules deriving Y for the grammar:

$S \rightarrow X\,\$$

$X \rightarrow X\ \texttt{id}\ Y$

$X \rightarrow \texttt{num}\ Y$

$Y \rightarrow +\ \texttt{id}\ Y$

$Y \rightarrow *\ \texttt{num}\ Y$

$Y \rightarrow \varepsilon$

`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 right associative,
- $\oplus$ is left associative,

Do the following:

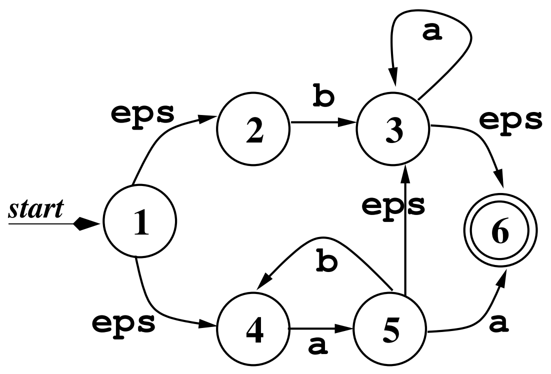
- **Rewrite the Grammar to be Unambiguous**:

- 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	sk_1 r1	...	sk_2 r2
...	...	...	...
state j	sk_3 r2	...	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) We want to extend FASTO with the second-order function `filter`, which receives as arguments a predicate (function) and an array of arbitrary element type and results in an array formed only by the input-array elements that succeed under the predicate. The type of `filter` is:

   `filter :` $(\alpha \rightarrow$ `bool`, $[\alpha]) \rightarrow [\alpha]$

   Write in the table below the type-checking pseudocode for `filter`, i.e., the high-level pseudocode for computing the result type of `filter(pred, arr_exp)` from the types of `pred` and `arr_exp`, together with whatever other checks are necessary. Assume that the function parameter of `filter` (i.e., `pred`) is passed only as a string denoting the function's name, i.e., do *NOT* consider the case of anonymous functions (lambda expressions). `vtable` and `ftable` denote the variable and function symbol tables, respectively. Try to stay close to the notation used in the textbook. Give self-explanatory error messages. The result of $Check_{Exp}$ is only the type of the `filter` expression.

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
`filter( pred, 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 := b / 5
 7:   b := b - v
 8:   w := a % 3
 9:   a := a - w
10:   GOTO begin
11:   LABEL end
12:   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`, and`w`.

4 Color the interference graph with 3 colors.

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