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, B2-2E14), 21st of January 2015

Exam Policy and Aids

- Exam lasts for four hours.
- The exam hand-out consists of 24 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 the book "Introduction to Compiler Design" by Torben Ægidius Mogensen, which the students may have printed themselves on regular paper (since an electronic edition is available).
- Students ***are also allowed*** to bring and freely use up to 8 sheets (16 pages) of printed or hand-written paper containing anything at all. In addition they are also allowed to bring an unlimitted number of blank paper.
- 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.
- 7 *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* 35 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.
- 5 *Longer-Answer Questions* 40 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 “Introduction to Compiler” (“Oversættere”)

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

Block 2, Winter 2014/15

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

1. I solemnly swear to fill in the evaluation form of the Compiler course this year and also that I am up to no good in doing so. True False
2. Register allocation is used to map an arbitrary number of program variables to a finite number of CPU registers. True False
3. The language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\}$, that is, the set of character sequences that start with n `a`’s, followed by n `b`’s, followed by n `c`’s, is representable with a context-free grammar (CFG). True False
4. The language formed by all keywords in the C programming language, {`"if"`,`"else"`,`"return"`,`...`} is representable via a Context-Free Grammar (CFG). True False
5. If α and β are regular expressions and $L(\alpha) = \{\mathtt{a}, \mathtt{c}\}$ and $L(\beta) = \{\mathtt{b}, \mathtt{d}\}$, then $L(\alpha \mid \beta) = \{\mathtt{ab}, \mathtt{ad}, \mathtt{cb}, \mathtt{cd}\}$, where $\alpha \mid \beta$ denotes alternation (union). True False
6. The regular expression $\mathtt{[a\text{-}zA\text{-}Z]}^+\mathtt{[0\text{-}9]}^*$ describes variable names that start with a letter and are followed by any sequence of interleaved letters and digits. True False
7. The intersection of two regular-expression languages is a regular expression. True False
8. With a NFA, the current state and input character determine *uniquely* the (transition to the) next state. True False
9. NFAs can accept languages that cannot possibly be accepted by DFAs. True False
10. DFA minimization can be used to test whether two regular expressions match the same language. True False
11. The language that consists of an unbounded, arbitrary number of nested matching open/close parentheses, i.e. { `()`, `(())`, $\cdots$}, can be matched with a regular expression. True False
12. Since it is possible to get runtime errors in SML, e.g. head of empty list, and since SML catches all types of runtime errors, it follows that SML is a *strongly-typed, dynamically-typed* language. True False
13. Without an intermediate language, translating m source languages to n different machine architectures would require implementing $m \times n$ translators. 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. (1pt) In the English language, the sentence "My dinner coookes mother not." contains

   a) a lexical error

   b) a syntax error

   c) a semantic error

   d) all of the above errors

2. (1pt) Assume an NFA $N = (S, s_0, F, T, \Sigma)$, where S is the set of states, $s_0 \in S$ is the starting state, $F \subseteq S$ is the set of accepting states, T is the set of transitions between states, and Σ is the alphabet. The ε-closure of a set of NFA states M, i.e. $\hat{\varepsilon}(M)$, is

   a) the set of *all characters* on which transitions from states in M are defined.
   $\hat{\varepsilon}(M)$ is computed by solving the equation $X = M \cup \{ c \in \Sigma \mid \exists s,t \in X : s^{\varepsilon}t \in M \}$

   b) the set of *all states* reachable from states in M via ε-transitions.
   $\hat{\varepsilon}(M)$ is computed by solving the equation $X = M \cup \{ s \in S \mid \exists t \in X : t^{\varepsilon}s \in T \}$,

   c) the set of *all transitions* reachable from states in M via ε-transitions.
   $\hat{\varepsilon}(M)$ is computed by solving the equation $X = M \cup \{ t \in T \mid \exists s \in X : s^{\varepsilon}t \in S \}$

   d) the set of *all non-final states* reachable from states in M via ε-transitions.
   $\hat{\varepsilon}(M)$ is computed by solving the equation $X = M \cup \{ t \in S - F \mid \exists s \in X : s^{\varepsilon}t \in T \}$,

3. (2pts) With respect to the tradeoff between implementing a DFA or an NFA,

   a) When translating an n-state NFA to a DFA, the *size of the DFA may explode exponentially*.

   c) A DFA tests membership in time proportional *only with input size* (not number of states).

   b) An NFA tests membership in time proportional with *its number of states and input size*.

   d) All of the above.

4. (2pts) If a formal *parameter* `p` is *not live* at the entry point of a function, then it means that

   a) the original value of `p` is never used, i.e., `p` is redefined before being used or is never used.

   b) parameter `p` may be used before being initialized.

   c) parameter `p` is only written inside the function and never read.

   d) all of the above.

5. (2pts) Using FASTO's current interpreter, the program

   ```
   fun int main() =if 5 < 3 then 'a' else 9
   ```

   results in:

   a) *value* `9` because the `else` branch is taken and `9` matches the return type of `main`.

   b) *a compile-time error* because the types of the `then` and `else` expressions differ.

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

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

6. (2pts) Callee-saves registers

   a) ideally hold variables that are not used in the caller after the function call.
   b) hold the actual arguments of the function call.
   c) are not saved by the callee if the callee does not use them.
   d) all of the above.

7. (2pts) Consider the SML program: `let val l = [] in if 3 < 4 then hd(l) else tl(l) end` where `hd` and `tl` return the head-element and the tail of the argument list, respectively.
   Note that both `l` and `tl(l)` have type `list(`α`)` and that `hd(l)` has type α. The program is type correct *if* the types of the `then` and `else` expressions unify, i.e., if α unifies with `list(`α`)`.
   Program compilation OR execution raise an error indicating that:

   a) α cannot possibly unify with `list(`α`)`; this is reported during compilation.
   b) the unification algorithm has resulted in an infinite recursion that was detected and reported as an error during compilation.
   c) `hd` is applied to an empty list; this happens during execution, after type-checking has unified α with `list(`α`)` (because a type variable always unifies with a type constructor).
   d) α cannot possibly unify with `list(`α`)`; this is reported during program execution.

III. Short Answers. Total 35 points.

1. (2pts) In the `Lexer.lex` file of the group project, explain the meaning of the regular expression `[' ' '\t' '\r']+` and its action, and why it is essential to be there? (Maximum 4 lines)

```
rule Token = parse
        [' ' '\t' '\r']+     { Token lexbuf }
       | ...
```

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

```
%token <(int*int)> PLUS IF THEN ELSE ...
%nonassoc ifprec ...
%left PLUS
%left TIMES
%type <Fasto.UnknownTypes.Exp> Exp
Exp : ...
    | Exp PLUS Exp { Plus ($1, $3, $2) }
    | IF Exp THEN Exp ELSE Exp %prec ifprec ...
```

a) Explain rule `Exp` $\rightarrow$ `Exp PLUS Exp`: (i) how is it disambiguated (what does `%left PLUS` 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 `ifprec` solve? (max 4 lines)

3. (3pts) For the alphabet $\Sigma = \{o, g\}$, write a NFA (or a regular expression) for arbitrary sequences of length n where n is divisible by 2, 3 or both.

4. (2pts) Consider the grammar $S \rightarrow \texttt{a}S\texttt{a} \mid \texttt{b}S\texttt{b} \mid \texttt{a} \mid \varepsilon$

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

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

5. (3pts) Consider the grammar $E \rightarrow E + E \mid E \texttt{**} E \mid \texttt{num}$, where `**` is considered a single terminal symbol (to power of). Given that: (i) `**` binds tighter than `+`, and that (ii) `+` is left associative, and that (iii) `**` is right associative, **Rewrite the Grammar to be Unambiguous**:

6. (4pts) The following pseudocode implements the interpretation of a function call (in FASTO)

```
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 => callFun(f, evargs, ftab)
5.          | NONE   => raise Error(..)
6.     end
7.   | ...
8. and callFun ( FunDec (fid, rtp, fargs, body, pdcl), aargs, ftab) =
9.     let val vtab' = bindParams (fargs, aargs)
10.        val res   = evalExp (body, vtab', ftab)
11.    in  if typeMatch (rtp, res) then res else raise Error(...)
12.    end
```

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

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

c) The shown implementation uses STATIC or DYNAMIC scoping? Circle one.

d) Explain in max 10 lines the modifications needed to implement the other form of scoping.

7. (3pts) Consider the C-like pseudocode below. What is printed under

```
void main() {                  void f(int a, int b) {
    int x = 3;                     a =   a + b;
    f(x,x);                        b = 2*a + b;
    print(x);                  }
}
```

Call By Value?	Call by Reference?	Call by Value Result?

8. (2pts) Under *dynamic scoping*, what does `main(3)` and `main(9)` print? (See C-like pseudocode below.)

```
void g() { print(x); }             void f(int x, int y) {
                                       if (y < 5) {           g(); }
void main(int y) {                     else       { int x = 9; g(); }
    f(3, y);                       }
}
```

9. (3pts) Do the types `(` α `* list(`α`) )` and `( list(char) *` β `)` unify? If they do what is the unified type, i.e., the most generic unifier (MGU)? The graph-representation of the two types is:

```
        *                         *
   _____|_____               _____|_____
   |         |               |         |
   V         V               V         V
alpha <--- list             list      beta
                             |
                             V
                            char
```

10. (3pts) 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` + 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`, z→`r_z`, q→`r_q`]`, translate the IL code below to MIPS code (with symbolic registers). Write your MIPS code on the right-hand side.

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

11. (3pts) Consider the MIPS code generator (at the page end) below for the logical-and operator, `&&`.

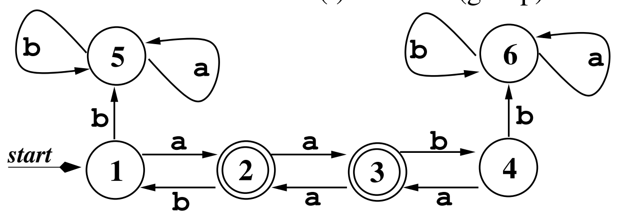
a) Explain in 2 lines the result of executing the following FASTO program:

```
fun bool main() = false && ((6/0) == 2)
```

b) Write on the right-hand side a different code generator for `&&`, which improves the provided one, and for which `main` returns `false`. Explain what it does in maximum 4 lines (space available on next page).

```
fun compileExp e vtable place =
case e of ...
| And (e1, e2, pos) =>
  let val t1 = newName "and_L"
      val t2 = newName "and_R"
      val code1 = compileExp e1 vtable t1
      val code2 = compileExp e2 vtable t2
  in code1 @
     code2 @
     [Mips.AND(place,t1,t2)]
  end
```

12. (4pts) Minimize the DFA below. (i) show all (group) tables ($\sim$5), and (ii) construct the minimized DFA.



IV. Longer Answers. Total 40 points. We leave two pages for the first two question, but it should **not** take that much writing.

1. (8pts)

Consider the following grammar:

$E \rightarrow E$ `num` `+`

$E \rightarrow E$ `num` `*`

$E \rightarrow$ `num`

a) Eliminate Left-Recursion of the provided grammar.

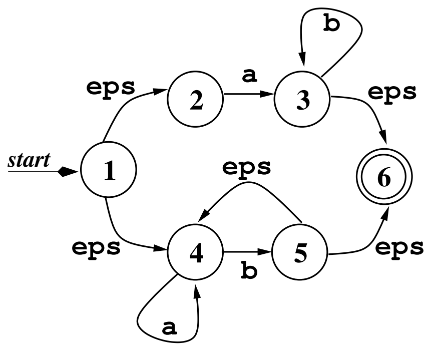
b) Do left factorization of the grammar obtained in a).

c) On the grammar obtained in b) compute:
the NULLABLE and FIRST sets for every production,
and the FOLLOW sets for every non-terminal.

d) Compute the Look-Ahead sets for every non-terminal.

e) Write a Recursive-Descent Parser in pseudocode.

2. (8pts) 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 (∼10) 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.)



3. (8pts) We want to extend FASTO with the second-order function `zipWith` which has the semantics:

`zipWith(f, {a`$_1$`,...,a`$_n$`}, {b`$_1$`,...,b`$_n$`}) = {f(a`$_1$`,b`$_1$`),..., f(a`$_n$`,b`$_n$`)}`

i.e., `zipWith` receives a function and two array-expression arguments and creates a new array by applying the function to pairs of same-index elements from the two arrays. `zipWith`'s type is:

`zipWith :` $((\alpha * \beta) \rightarrow \gamma,\ [\alpha],\ [\beta]) \rightarrow [\gamma]$

Write the type checking for `zipWith(f, exp1, exp2)` **below**, i.e., the high-level pseudocode of how to compute the result type of `zipWith(f, exp1, exp2)` from the types of `f`, `exp1` and `exp2`, together with whatever other checks are necessary. (`vtable` and `ftable` denote the variable and function symbol tables, respectively.) Try to stay close to the notation used in the textbook.

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
`zipWith(f, exp1, exp2)`	

4. (8pts) **This task refers to copy/constant propagation and constant folding.**

```
datatype Propagatee = ConstProp of Value
                    | VarProp   of string

fun copyConstPropFoldExp vtable e =
case e of ...
1.  | Let (Dec (name, e, decpos), body, pos) =>
2.      let val e'      = copyConstPropFoldExp vtable e
3.          val vtable' =
4.            case e' of
5.              (Var      (newname, _)) => SymTab.bind name (VarProp newname) vtable
6.            | (Constant (value,   _)) => SymTab.bind name (ConstProp value) vtable
7.            |  _                      => SymTab.remove name vtable

8.          val body' = copyConstPropFoldExp vtable' body
9.      in  Let (Dec (name, e', decpos), body', pos) end
```

a) The code above shows the copy/constant propagation implementation for a `let`-binding expression. **Explain lines 5,6,7: why is `vtable` updated and what it contains**. Fill in below:

line 2: subexpression `e` is optimized under the current symbol table, resulting in `e'`.

line 3: a new symbol table is created as follows:

line 5: If `e'` is a

line 6: If `e'` is a

line 7: Otherwise the binding of `name` (if any) is removed from the symbol table because the following nasty thing might happen:

line 8 Finally, the `body` of the `let` is optimized under the new symbol table.

b) **Constant/copy propagation for variables; Fill in the blanks at lines 2 and 3 below:**

```
fun copyConstPropFoldExp vtable e = case e of ...
  | Var (name, pos) =>
1.    (case SymTab.lookup name vtable of


2.           SOME (VarProp newname) => ................................


3.         | SOME (ConstProp value) => ................................
4.         | NONE                   => Var (name, pos))
```

c) Write below the implementation of constant folding for a `Divide` expression, i.e., `e1 / e2`, and explain briefly (in max 5 lines):

```
fun copyConstPropFoldExp vtable e =
case e of ...
  | Divide (e1, e2, pos) =>
      let val e1’ = copyConstPropFoldExp vtable e1
          val e2’ = copyConstPropFoldExp vtable e2
      in case (e1’, e2’) of













           | _ => Divide (e1’, e2’, pos)
      end
```

5. (8pts) Liveness-Analysis and Register-Allocation Exercise. Given the following program:

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

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 graph for `a`, `b`, `c`, `t`, `v`.

4 Color the interference graph with 4 colors.

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