Cosmin Oancea
Answers and Marking Scheme.

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 — either one 1pt
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 $\{a^n b^n c^n \mid n \in \mathbb{N}\}$, 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 $L(\alpha) = \{a, c\}$ and $L(\beta) = \{b, d\}$, then $L(\alpha \mid \beta) = \{ab, ad, cb, cd\}$, where $\alpha \mid \beta$ denotes alternation (union). — True / (False)
6. The regular expression `[a-zA-Z]+[0-9]*` 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. {(), (()), ···}, 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 }
      | ...
```

(1pt) The regular expression matches a sequence of one or more consecutive "white spaces". The action ignores the white space (used to separate tokens)
(1pt) and calls the lexer (recursively) on the rest of the character stream so that all tokens are identified (otherwise only one token is identified)

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` → `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)

- Each terminal and non terminal is declared to carry info of some type, e.g. nonterminal "Exp" carries an AbSyn expression (Fasto.UnknownType.Exp)
- Each production builds a piece of the AbSyn type associated to its left-hand-side nonterminal from the info associated to the right-hand-side symbols. $n refers to the info of symbol number "n" in the production.

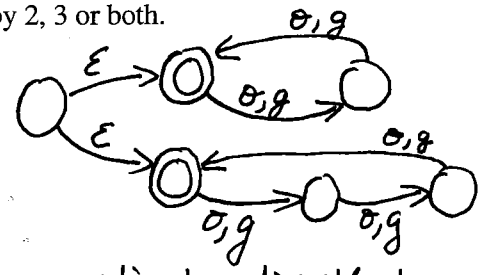
(1pt) As such ~~parsing~~ Exp PLUS Exp builds a Plus (AbSyn) expression from the two subexpressions ($1 and $3) and the line-column position of terminal PLUS ($2)

(1pt)
- Grammar is disambiguated by declaring operator PLUS left associative ((E+E)+E) and by assigning PLUS a lower precedence than multiplication.

b) Explain what does the use of `ifprec` solve? (max 4 lines)

(1pt)
- solves the ambiguity introduced by an if expression, e.g. "if cond then 1 else 2+3" can be parsed as "(if cond then 1 else 2) + 3 OR as "(if cond then 1 else (2+3))"
- ifprec set as lowest precedence indicates that the else-expression should be prolonged as much as possible (shift over reduce).

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.



$((o|g)(o|g))^* \mid ((o|g)(o|g)(o|g))^*$

1 pt for the NFA corresponding to divisible by 2 (if it can be identified)
1 pt for the NFA —"— —"— 3 (if it can be identified)
1 pt for putting the 2 NFA (regexp) together correctly.

4. (2pts) Consider the grammar $S \rightarrow$ aSa | bSb | a | ε

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:
$S \Rightarrow$ $a\underline{S}a \Rightarrow ab\underline{S}ba \Rightarrow abb\underline{S}bba \Rightarrow abbabba$

5. (3pts) Consider the grammar $E \rightarrow E + E \mid E$ ** $E \mid$ 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**:

$E \rightarrow E+T \mid T$

$T \rightarrow F ** T \mid F$

$F \rightarrow num$

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.

1 pt

- line 2: the actual arguments are evaluated
- line 3: a lookup for function named "fid" is performed on the function symbol table
- line 4: lookup succeeds and returns the function declaration ("f"), "callFun" evaluates the function call (on the already evaluated actual params).

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

1 pt

- line 9: a new variable symbol table is created that contains only the bindings of formal-param names with the actual-arg values
- line 10: the body of the function is evaluated on the created vtable
- line 11: checks that the result of the body evaluation matches the return type of the function.

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

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

- "callFun" should receive an extra parameter: the "current vtable" at the point of the call
- the formal-to-actual argument bindings should be added to the "current vtable" (rather than to an empty one).

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

```
void main() {
    int x = 3;
    f(x,x);
    print(x);
}

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

either 15 OR 6 gets (1pt)

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

a = ~~A~~(=3); b = ~~A~~ (=3)
a = a+b
b = 2*a+b
No modification to x
PRINTS 3 (1pt)
	x = x+x (=6)
	x = 2*x+x (=18)
	PRINTS 18
	(1pt)
		a = x (=3); b = x (=3);
		a = a+b (=6)
		b = 2*a+b (=2·6+3 = 15)
		x=a; x=b; OR x=b; x=a;
		PRINTS 15 OR 6 (depending on copy-out order)

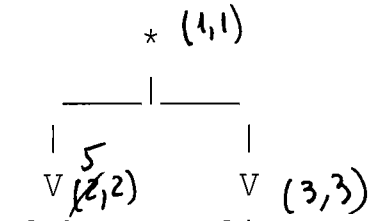
8. (2pts) Under *dynamic scoping*, what does `main(3)` and `main(9)` print? (See C-like pseudocode below.)

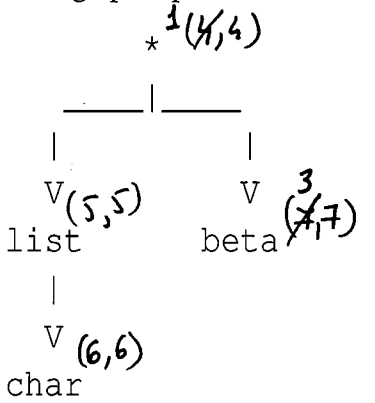
```
void g() { print(x); }

void main(int y) {
    f(3, y);
}

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

1pt main (3) prints 3

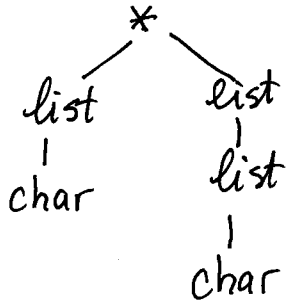
1pt main (9) prints 9

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:





Either



OR (list(char) * list(list(char)))

OR α = list(char) and β = list(list(char))

gets all (3pts)

• partial marks can be given if unification was tried. (but incorrect from a point on).

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:
```

addi r_q, r_q, 4
lw r_x, 4(r_q)
bne r_x, r_z, lab2
lab_1:

Suggested: for every mistake subtract 1pt.

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)
```

(1pt) → the implementation of && is not short circuited hence the evaluation of 6/0 will result in a runtime error.

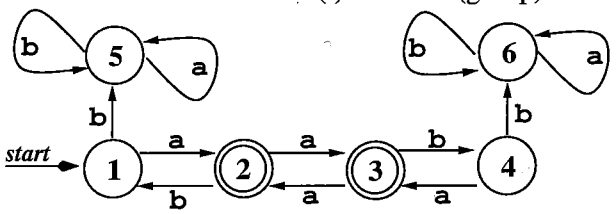
(2pts) b) Write on the right-hand side a different code generator for &&, which improves the provided one, and for which main returns false. Explain what it does in maximum 4 lines (space available on ~~next page~~).

```
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
```

```
fun compileExp e vtable place =
  case e of ...
  | And (e1, e2, pos) =>
    let val lab = newName "sc_lab"
        val code1 = compileExp e1 vtable place
        val code2 = compileExp e2 vtable place
    in code1 @
       [Mips.BEQ (place, "0", lab)] @
       code2 @
       [Mips.Label (lab)]
    end
```

- both subexpressions are compiled such as results are in "place"
- the resulting code is separated by an "beq" branch instr that guarantees that if the first subexpression evaluates to false ("0") then the code of the second subexpression is not execute (jump after it).
- "place" will be true ("1") only if both subexpressions evaluate to true ("1").

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



$G_1 = \{2,3\}$, $G_2 = \{1,4,5,6\}$.

G_2	a	b
1	G_1	G_2
4	G_1	G_2 ← split
5	G_2	G_2
6	G_2	G_2

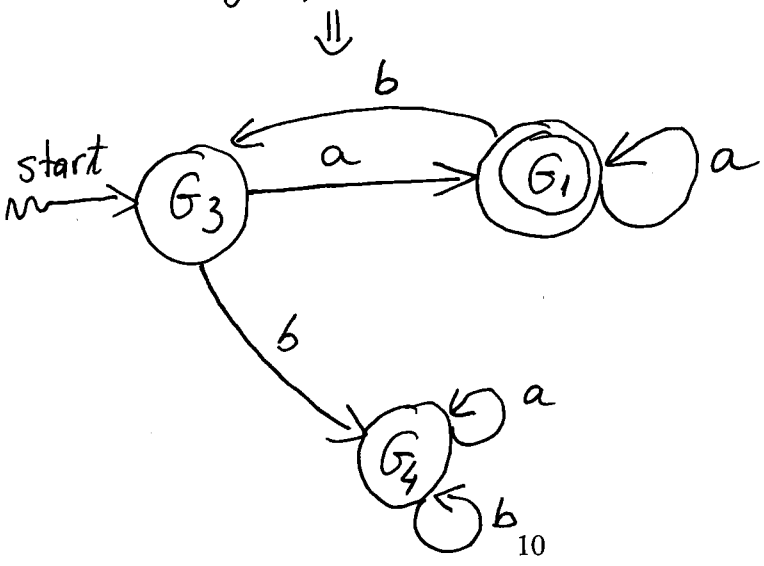
$G_1 = \{2,3\}$, $G_3 = \{1,4\}$, $G_4 = \{5,6\}$

G_1	a	b
2	G_1	G_3
3	G_1	G_3

G_3	a	b
1	G_1	G_4
4	G_1	G_4

G_4	a	b
5	G_4	G_4
6	G_4	G_4.

all groups consistent



(G_4 is dead and removing it is OK, i.e. should not loose mark if G_4 not shown)

(partial marks if they "seem" to know what they are doing).

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.

a.) (2pts)
$E \rightarrow num\ E'$
$E' \rightarrow num + E'$
$E' \rightarrow num * E'$
$E' \rightarrow \varepsilon$

b) (1pt)
$S \rightarrow E\$$ (0)
$E \rightarrow num\ E'$ (1)
$E' \rightarrow num\ y$ (2)
$E' \rightarrow \varepsilon$ (3)
$y \rightarrow + E'$ (4)
$y \rightarrow * E'$ (5).

c.) (2pts)
NULLABLE(ε) = True
NULLABLE(E') = True
Nullable of anything else is False

FIRST(num E) = {num}
FIRST(num y) = {num}
FIRST(+ E') = {+}
FIRST(* E') = {*}
FIRST(E) = {num}
FIRST(E') = {num}
FIRST(y) = {+, *}

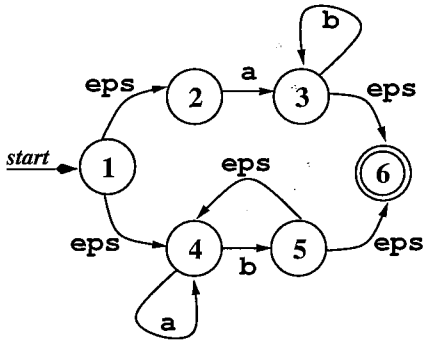
{$} ⊆ FOLLOW(E)
FOLLOW(E) ⊆ FOLLOW(E')
FOLLOW(E') ⊆ FOLLOW(y)
FOLLOW(y) ⊆ FOLLOW(E')
⇓
{$} = FOLLOW(E) = FOLLOW(E') = FOLLOW(y).

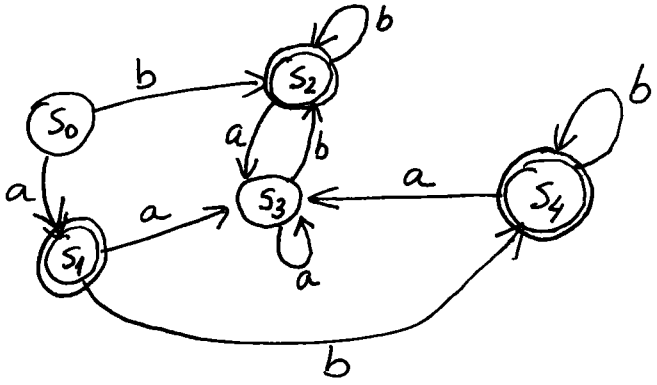
c.) (1pt)
la($S \rightarrow E\$$) = {num}
la($E \rightarrow num\ E'$) = {num}
la($E' \rightarrow num\ y$) = {num}
la($E' \rightarrow \varepsilon$) = FOLLOW($E'$) = {$}
la($y \rightarrow + E'$) = {+}
la($y \rightarrow * E'$) = {*}.

(2 pts)

```
fun parseS () = if next = num then print(0); parseE(); match($);
                else ERROR(...)

fun parseE() = if next = num then print(1); match(num); parseE'();
               else ERROR(...)

fun parseE'() =
    if next = num then print(2); match(num); parseY();
    else if next = '$' then print(3)
    else ERROR(..);

fun parseY() =
    if next = '+' then print(4); match('+'); parseE'();
    else if next = '*' then print(5); match('*'); parseE'();
    else ERROR(...);
```

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.)





2pts

$\hat{\varepsilon}(\{1\}) = \{1,2,4\} = S_0$

$move(S_0, a) = \hat{\varepsilon}(\{3\}) = \{3,4,6\} = S_1$ Final

$move(S_0, b) = \hat{\varepsilon}(\{5\}) = \{5,4,6\} = S_2$ Final

$move(S_1, a) = \hat{\varepsilon}(\{4\}) = \{4\} = S_3$

$move(S_1, b) = \hat{\varepsilon}(\{3,5\}) = \{3,5,6,4\} = S_4$ Final

$move(S_2, a) = \hat{\varepsilon}(\{4\}) = S_3$

$move(S_2, b) = \hat{\varepsilon}(\{5\}) = S_2$

$move(S_3, a) = \hat{\varepsilon}(\{4\}) = S_3$

$move(S_3, b) = \hat{\varepsilon}(\{5\}) = S_2$

$move(S_4, a) = \hat{\varepsilon}(\{4\}) = S_3$

$move(S_4, b) = \hat{\varepsilon}(\{3,5\}) = S_4$

6pts

partial marks in abundance, i.e. ~~me~~ if an error try to see if ~~assu~~ the rest of the answer is correct modulo the error.

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

`zipWith(f, {a1,...,an}, {b1,...,bn}) = {f(a1,b1),..., f(an,bn)}`

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, expl, exp2)` **below**, i.e., the high-level pseudocode of how to compute the result type of `zipWith(f, expl, exp2)` from the types of `f`, `expl` 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, expl, exp2)`

```
tarr1 = Check_Exp (exp1, vtable, ftable)                    (1 pt)
tarr2 = Check_Exp (exp2, vtable, ftable)                    (1 pt)
tel1 = case tarr1 of                                        (1 pt)
         [t1] => t1
       | other => ERROR("array arg not an array!")
tel2 = case tarr2 of                                        (1 pt)
         [t2] => t2
       | other => ERROR("array arg not an array!")
tfun = lookup(ftable, name(f))                              (1 pt)
case tfun of
     unbound => ERROR("fun not found!")
   | ((tin1 * tin2) -> tout) =>
         if tin1 = tel1 andalso                             (1 pt)
            tin2 = tel2                                     (1 pt)
         then [tout]
         else ERROR("fun type does not match array-elem type")
   | otherwise => ERROR("fun does not receive exactly two args")   (1 pt)
```

] denoted [t] to be the type of an array whose elements have type t

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 variable then bind it to the symbol table as a variable propagatee (because it corresponds to a copy-propagation statement "let x = y in.."

1pt

line 6: If `e'` is a constant (value) then bind it to the symbol table as a constant propagatee (because it corresponds to a constant-propagation stmt: "let x = 5 in"

1 pt

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

Eg:
1. let y = x in
2. let y = x*x in
3. y+2

When processing the let-binding at line 2, if the previous binding of y, i.e., y ↦ x, is not removed from vtable then the result "y+2" will be transformed to "x+2", which is incorrect. The problem is that created by ~~allowing~~ the first decl of y being overshadowed by the second one (and the second does not add a binding to vtable because it is not a copy/ct statement).

1 pt

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

Acceptable answer also: the problem corresponds to shadowing variable names, hence ~~the~~ an "invisible" overshadowed binding needs to be removed from vtable.

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) => ..... Var (newname, pos)      1pt
3.        | SOME (ConstProp value) => ..... Constant (value, pos)  1pt
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):

3 pts.

```
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
(1)       (_, Constant(IntVal 0,_)) => ERROR ("division by 0!")
(2)     | (Constant(IntVal 0,_), _) => e1'
(3)     | (_, Constant(IntVal 1,_)) => e1'
(4)     | (Constant(k1,_), Constant(k2,_)) => Constant(IntVal(k1 div k2), pos)

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

(1) division by constant value 0 ⇒ error is statically reported

(2) 0/x is simplified to 0

(3) x/1 is simplified to x

(4) if both e_1' and e_2' are constants then division is performed statically and result is wrapped into an AbSyn constant value.

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. — 1 pt

2 Compute **in** and **out** sets for every instruction; stop after two iterations. — 3 pts

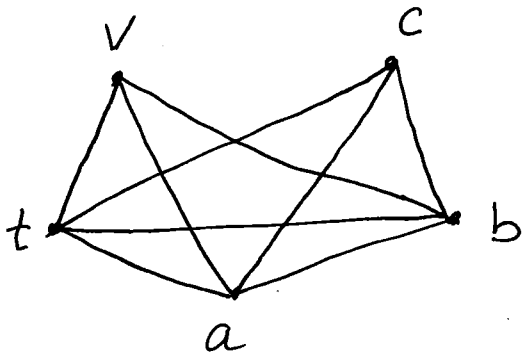
3 Draw the interference graph for a, b, c, t, v. — → 2 pts

4 Color the interference graph with 4 colors. } → 2 pts

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

i	succ	gen	kill	out1	in1	out2	in2
1	2			a,b,c	a,b,c	a,b,c	a,b,c
2	3,10	a		a,b,c	a,b,c	a,b,c	a,b,c
3	4			a,b,c	a,b,c	a,b,c	a,b,c
4	5	b	t	a,b,c,t	a,b,c	a,b,c,t	a,b,c
5	6	c	v	a,b,t,v	a,b,c,t	a,b,t,v	a,b,c,t
6	7	b,v	b	a,t	a,b,t,v	a,b,t	a,b,t,v
7	8	t	c	a	a,t	a,b,c	a,b,t
8	9	a	a		a	a,b,c	a,b,c
9	1					a,b,c	a,b,c
10	11			b	b	b	b
11	-	b			b		b

$$\begin{cases} in[i] = gen[i] \cup (out[i] \setminus kill[i]) \\ out[i] = \bigcup_{k \in succ[i]} in[k] \end{cases}$$



Stmt	Var	Interferes with
4	t	a, c
5	v	a, b, t
6	b	a, t
7	c	a, b
8	a	b, c

Node	Neighors	Color
a		1
b	a	2
t	a, b	3
c	a, b, t	4
v	a, b, t	4

Colorable with 4 colors

As always, try to give partial marks across a mistake if
- ~~partial mark~~ the rest is consistent (modulo the mistake).