Cosmin Oancea

Answers and Marking Scheme.

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.

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 2015/16

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

1. The language $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$, 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 }
        | ...
```

1pt { The regular expression matches a comment, i.e., sequence of chars that starts with "//" and ends just before the start of a new line. The action ignores the
1pt { comment and calls the lexer recursively on the rest of the char stream so that all tokens are identified (otherwise tokenization ends there).

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 { Plus ($1, $3, $2) }
    | LET ID EQ Exp IN Exp %prec letprec ...
```

typo: should be "Or" instead of "Plus"

a) Explain rule `Exp → 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)

Each production builds a piece of the AbSyn associated to its left-hand nonterminal from the info associated to the right-hand side symbols.
1pt { For example "Exp → Exp OR Exp" builds a ~~Exp~~ "Fasto.Unknown.Exp" using ~~the~~ its constructor "Plus" (typo: should be "Or") from the info of the two subexpressions ($1 and $3) and the position of "OR" ($2). ($n refers to the info carried by the n^th symbol in the production).
1pt { Grammer is disambiguated by declaring operator "OR" left associative ((E OR E) OR E) and by assigning OR lower precedence than AND.

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

1pt {
- Solves the ambiguity of a let expression, for example let x=3 in x+5 can be parsed as { (let x=3 in x)+5 (i) OR (let x=3 in (x+5)) (ii)
- letprec is set as the lowest precedence operator indicating to shift (on +) rather than to reduce (the let), hence (ii).

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.

1pt { `0 | [1-9][0-9]*0 |` ← multiples of 10

2pts { `3 [^ 3]* 3 [^ 3]* |` ← two 3s starting with 3

`[1-2,4-9][^ 3]* 3 [^ 3]* 3 [^ 3]*` ← two 3s not starting with 3

Subtract one point if they assume number can start with a number of zeroes, e.g. 000123

4. (2pts) Consider the grammar

$S \rightarrow aSb \mid B$

$B \rightarrow bB \mid b$

a) Can *aaabbb* be derived from *S*? YES (NO)

If it can, show the derivation on the line below:

$S \Rightarrow$

1pt

b) Can *aabbbb* be derived from *S*? (YES) NO

If it can, show the derivation on the line below:

$S \Rightarrow aSb \rightarrow aaSbb \rightarrow aaBbb \rightarrow aabBbb \rightarrow aabbbb$

1pt

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
```

typo: should be vtab''

if the explanation is consistent do NOT subtract marks.

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

2 pt
line 2: the actual arguments are evaluated
line 3: look up performed on the function symbol table
line 4: look up succeeds and returns the function declaration AbSyn of "f" then "callFun" evaluates the function's body on the evaluated params

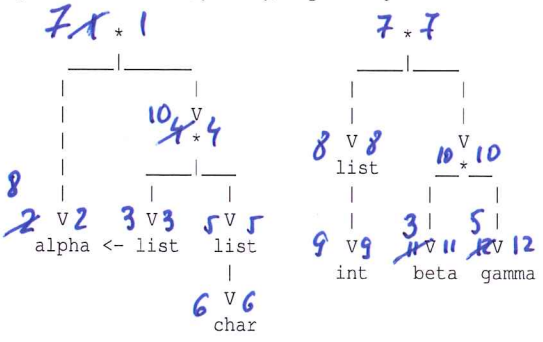
b) Explain in maximum 4 lines what lines 9, 10, 11 and 12 do.

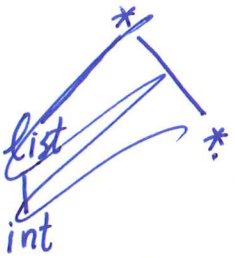
2 pt
line 9: a new variable symbol table created by binding formal arg names with actual args values
line 10: vtab" created by appending vtab' on top of the current vtab
line 11: body of the function is evaluated (on vtab' or vtab")
line 12: checks function's return type matches the result of body.

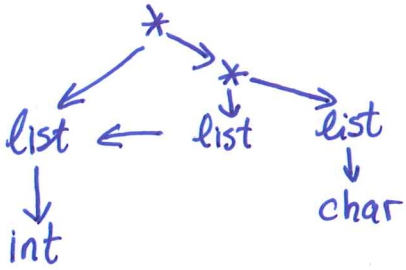
c) The shown implementation uses STATIC or DYNAMIC scoping? Circle one. Explain in max 2 lines the modifications needed to implement the other form of scoping.

2 pt
Due to typo:
IF DYNAMIC was chosen ⇒ pass SymTab.empty() as argument to callFun with Vtable in line 4 (instead of vtab)
IF STATIC was chosen ⇒ pass vtab" instead of vtab' as argument in line 11

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.







1 pt α ≡ list(int)
1 pt β ≡ list(list(int))
1 pt γ ≡ list(char)
1 pt unified ≡ list(int) * (list(list(int)) * list(char))

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:
```

not last use of y

addi r_y, r_q, 4
sw r_x, 0(r_y)
} 2 pts

bne r_x, r_y, lab_2
lab_1:
} 1 pt

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;
}
```

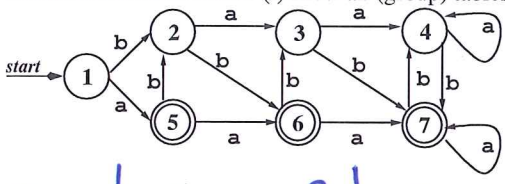
Label $loop_{start}$:
$t_1 := r_y$
$t_2 := 0$
Label lab_1: IF $t_1 > t_2$ THEN lab_1 ELSE $loop_{end}$
$t_3 := r_x$
$t_4 := 100$
IF $t_3 < t_4$ THEN lab_2 ELSE $loop_{end}$
Label lab_2:
continued there

$t_5 := r_y$
$t_6 := 1$
$r_y := t_5 - t_6$
$t_7 := r_x$
$t_8 := 2$
$r_x := t_7 + t_8$
GOTO $loop_{start}$
Label $loop_{end}$:

Suggested marking: 1 pt for correct arithmetic semantics
1 pt for correct use of temporaries, i.e., following book algorithm for translation
2 pts for correct shortcircuiting, i.e., the two IFs and the loop

-1 pt if source-lang variables are used (instead of the vtab translation).

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

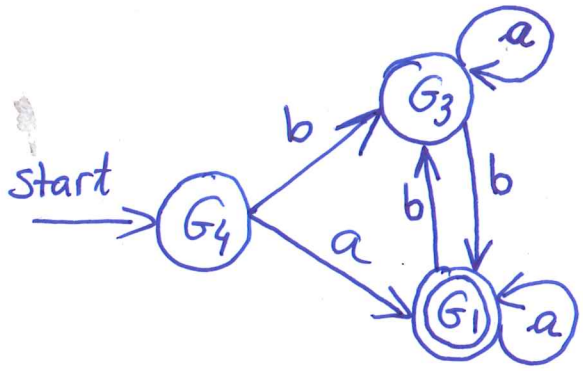


G_2	a	b
1	G_1	G_2
2	G_2	G_1
3	G_2	G_1
4	G_2	G_1

G_1	a	b
5	G_1	G_3
6	G_1	G_3
7	G_1	G_3

G_3	a	b
2	G_3	G_1
3	G_3	G_1
4	G_3	G_1

G_4	a	b
1	G_1	G_3



Marking: 3 pts for correct tables
2 pts for correct minimized DFA (graph) or consistent with tables.

10. (2pts) Eliminate the left-recursion of the grammar bellow (and write down the resulted equivalent grammar):

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow \texttt{num}$

1pt $E \rightarrow \texttt{num}\ E' \mid (E)\ E'$
1pt $E' \rightarrow +\ E\ E' \mid *\ E\ E' \mid \varepsilon$

11. (5pts) Compute $\texttt{Follow}(Y)$ and the lookahead sets of the grammar rules deriving Y for the grammar:

$S \rightarrow X\ \$$
$X \rightarrow (\ X\ )\ Y$
$X \rightarrow \texttt{id}\ Y$
$Y \rightarrow \texttt{num} + Y$
$Y \rightarrow \texttt{id} * Y$
$Y \rightarrow \varepsilon$

'$' $\in$ FOLLOW(X)
')' $\in$ FOLLOW(X)
FOLLOW(X) $\subseteq$ FOLLOW(Y)

$\Downarrow$

2pts $\texttt{Follow}(Y) = \{\ '\$',\ ')'\ \}$

1pt $\texttt{LookAhead}(Y \rightarrow \texttt{num} + Y) = \{\ \texttt{num}\ \}$

1pt $\texttt{LookAhead}(Y \rightarrow \texttt{id} * Y) = \{\ \texttt{id}\ \}$

1pt $\texttt{LookAhead}(Y \rightarrow \varepsilon) =$ FOLLOW(Y) $= \{\ '\$',\ ')'\ \}$

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:

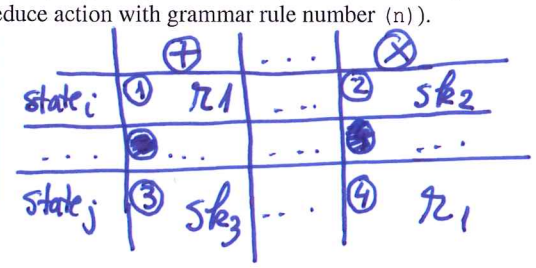
- **Rewrite the Grammar to be Unambiguous**:

2 pts

$E \rightarrow T \oplus E \mid T$
$T \rightarrow T \otimes F \mid F$
$F \rightarrow$ num

- 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 rn 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



1 pt ① corresponds to a ⊗ b ⊕ c : since ⊗ has higher precedence than ⊕ reduce (on rule(1)) is chosen over shift

1 pt ② corresponds to a ⊕ b ⊗ c : since ⊗ has higher precedence shift is chosen over reduce (on rule2)

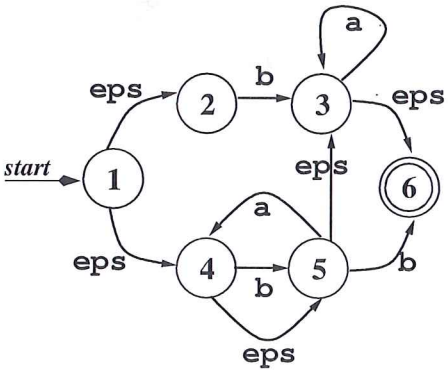
1 pt ③ corresponds to a ⊕ b ⊕ c : since ⊕ is right associative shift is preferred to reduce

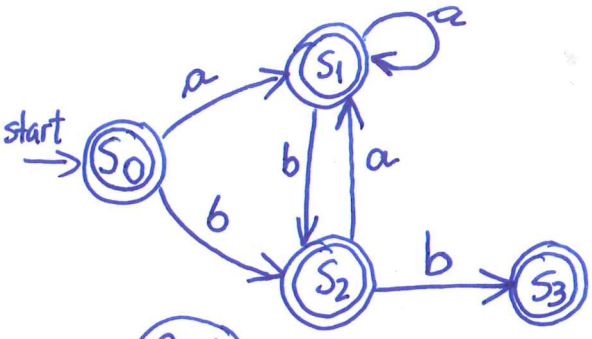
1 pt ④ corresponds to a ⊗ b ⊗ c : since ⊗ is left associative reduce ~~is preferred~~ on rule (1) is chosen over shift.

Typo: Total 30points.

IV. Longer Answers. Total 40 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.)





2pts for DFA consistent with the computed move function

8pts

$\hat{\varepsilon}(\{1\}) = \{1, 2, 4, 5, 3, 6\} = S_0$ final

move $(S_0, a) = \hat{\varepsilon}(\{3, 4\}) = \{3, 4, 6, 5\} = S_1$ final

move $(S_0, b) = \hat{\varepsilon}(\{3, 5, 6\}) = \{3, 5, 6\} = S_2$ final

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

move $(S_1, b) = \hat{\varepsilon}(\{5, 6\}) = \{5, 6, 3\} = S_2$.

move $(S_2, a) = \hat{\varepsilon}(\{3, 4\}) = S_1$

move $(S_2, b) = \hat{\varepsilon}(\{6\}) = \{6\} = S_3$ final

move $(S_3, a) = \emptyset$

move $(S_3, b) = \emptyset$

-1pt if final states are wrong.

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

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

1 pt $\forall \alpha \left(((\alpha * \alpha) \rightarrow \alpha) * \alpha * [\alpha] \right) \rightarrow \alpha$ 1 pt

(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)`	t_n = Check_Exp(n_exp, vtable, ftable) } 1 pt
	t_arr = Check_Exp(arr_exp, vtable, ftable) }
	t_el = case t_arr of } 1 pt
	  Array(t_1) ⇒ t_1
	  | other ⇒ Error("array arg not array")
	t_f = lookup(ftable, name(f)) } 1 pt
	case t_f of
	  unbound ⇒ Error("function not found") } 1 pt
	| (t_{in1}, t_{in2}) → t_{out} ⇒
	  if $t_{in1} = t_{in2}$ and $t_{in2} = t_{out}$ } 1 pt
	  then if t_n = t_{in1} } 1 pt
	  then if t_n = t_el then t_n
	  else Error("array elem type not matches accum type") } 1 pt
	  else Error("accum type not matches funarg type") } 1 pt
	  else Error("function not of type $\alpha * \alpha \rightarrow \alpha$") } 1 pt
	| other ⇒ Error("function does not receive exactly two arguments")

9 pts

They can use [t] for an array type or whatever other notation

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. 1 pt

2 Compute **in** and **out** sets for every instruction; stop after two iterations. 5 pts

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

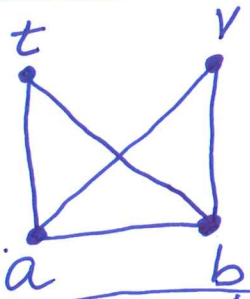
4 Color the interference graph with 3 colors. } 2 pts

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	a,b	a,b	a,b
2	3,9	a		a,b	a,b	a,b	a,b
3	4			a,b	a,b	a,b	a,b
4	5	a,b	t	a,b,t	a,b	a,b,t	a,b
5	6	b,t	b	a	a,b,t	a,b	a,b,t
6	7	a	v	a,v	a	a,b,v	a,b
7	8	a,v	a		a,v	a,b	a,b,v
8	1					a,b	a,b
9	10			b	b	b	b
10	-	b			b		b

stmt	kill	interferes with
4	t	a, b
5	b	a
6	v	a, b
7	a	b

~~Node~~



Node	Neighbors	Color
a		1
b	a	2
v	a, b	3
t	a, b	3

If they do not start with t or v (on stack) & then − 1pt.