Programming Language Technology

Exam, 13 January 2020 at 08.30 – 12.30 in M

Course codes: Chalmers DAT151, GU DIT231. As re-exam, also DAT150 and DIT230.
Exam supervision: Andreas Abel (+46 31 772 1731), visits at 09:30 and 11:30.

Grading scale: Max = 60p, VG = 5 = 48p, 4 = 36p, G = 3 = 24p.
Allowed aid: an English dictionary.
Exam review: 24 January 2019 13.30-15.00 in EDIT meeting room *Analysen* (3rd floor).

Please answer the questions in English.

Question 1 (Grammars): Write a labelled BNF grammar that covers the following kinds of constructs of C/C++:

- Program: `int main()` followed by a block
- Block: a sequence of statements enclosed between `{` and `}`
- Statement:
  - block
  - initializing variable declaration, e.g., `int x = e;`
  - statement formed from an expression by adding a semicolon `;`
  - `if` statement with `else`
- Expression:
  - boolean literal `true` or `false`
  - integer literal
  - function call with a list of comma-separated arguments
  - post-increment of an identifier, e.g., `x++`
  - addition (+), left associative
  - parenthesized expression
- Type: `int` or `bool`

You can use the standard BNFC categories `Integer` and `Ident`, the `coercions` pragma, and list categories via the `terminator` and `separator` pragmas. An example program is:

```
#include <stdio.h>
#define printInt3(e1,e2,e3) printf("%d %d %d\n",e1,e2,e3)
int main () {
  int x = 8;
  if (true) {
    printInt3 (x++, 10 + x++, x++ + 19);
  } else bool b = false;
}
```

Herein, lines starting with `#` are comments, but you do can ignore comments in your grammar.
(10p)

CLARIFICATION:

1. Said *constructs of C*, but boolean literals were added in C/Java only.
2. Said *Lines starting with # are comments* but did not include the `comment` pragma in the list of allowed BNFC features.
3. Includes function calls but not function definitions, which confused at least one student.

SOLUTION:

```
Program.  Prg   ::= "int" "main" "(" ")" "{" [Stm] "}"  ;

SDecl.    Stm   ::= Type Ident "=" Exp ";"              ;
SExp.     Stm   ::= Exp ";"                             ;
SIfElse.  Stm   ::= "if" "(" Exp ")" Stm "else" Stm     ;
SBlock.   Stm   ::= "{" [Stm] "}"                       ;

terminator Stm ""                                       ;

ETrue.      Exp1  ::= "true"                            ;
EFalse.     Exp1  ::= "false"                           ;
EInt.       Exp1  ::= Integer                           ;
EPostIncr.  Exp1  ::= Ident "++"                        ;
ECall.      Exp1  ::= Ident "(" [Exp] ")"               ;
EPlus.      Exp   ::= Exp "+" Exp1                      ;

separator Exp ","                                       ;

TInt.     Type  ::= "int"                               ;
TBool.    Type  ::= "bool"                              ;

coercions Exp 1                                         ;

comment "#"                                             ;
```

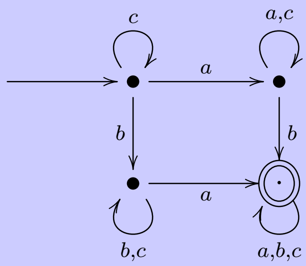
Question 2 (Lexing): An *acceptable password* be a sequence of characters that contains at least one digit and at least one special character. Our alphabet be $\Sigma = \{a, b, c\}$ where a stands for digits, b for special characters, and c for other characters (like letters).

1. Give a regular expression for acceptable passwords.
2. Give a deterministic finite automaton for acceptable passwords with no more than 8 states.

Remember to mark initial and final states appropriately. (4p)

SOLUTION:

1. RE: $\Sigma^*(a\Sigma^*b + b\Sigma^*a)\Sigma^*$
2. DFA:



CLARIFICATION: The original formulation *at least one digit and one special character* was ambiguous. Fixed. Announced (blackboard) during exam. Solutions that insist on *exactly one special character* should be accepted as well.

Question 3 (LR Parsing): Consider the following labeled BNF-Grammar (written in `bnfc` syntax). The starting non-terminal is `S`.

```
Seq.   S ::= P M     ;

Plus.  P ::= P A "+" ;
None.  P ::=         ;

Minus. M ::= A "-" M ;
Done.  M ::= A       ;

X.     A ::= "x"     ;
Y.     A ::= "y"     ;
```

Step by step, trace the shift-reduce parsing of the expression

```
x + y - x
```

showing how the stack and the input evolve and which actions are performed. (8p)

SOLUTION: The actions are `shift`, `reduce with` rule(s), and `accept`. Stack and input are separated by a dot.

```
        . x + y - x     -- reduce with rule None
P       . x + y - x     -- shift
P x     . + y - x       -- reduce with rule X
P A     . + y - x       -- shift
P A +   . y - x         -- reduce with rule Plus
P       . y - x         -- shift
P y     . - x           -- reduce with rule Y
P A     . - x           -- shift
P A -   . x             -- shift
P A - x .               -- reduce with rule X
P A - A .               -- reduce with rule Done
P A - M .               -- reduce with rule Minus
P M     .               -- reduce with rule Seq
S       .               -- accept
```

Question 4 (Type checking and evaluation):

1. Write syntax-directed *type checking* rules for the *statement* forms and blocks of Question 1. The form of the typing judgements should be $\Gamma \vdash s \Rightarrow \Gamma'$ where s is a statement or list of statements, Γ is the typing context before s, and Γ' the typing context after s. Observe the scoping rules for variables! You can assume a type-checking judgement $\Gamma \vdash e : t$ for expressions e.

   Alternatively, you can write the type checker in pseudo code or Haskell (then assume `checkExpr` to be defined). In any case, the typing environment must be made explicit. (6p)

SOLUTION: We use a judgement $\Gamma \vdash s \Rightarrow \Gamma'$ that expresses that statement s is well-formed in context Γ and might introduce new declarations, resulting in context Γ'.

A context Γ is a stack of blocks Δ, separated by a dot. Each block Δ is a map from variables x to types t. We write $\Delta, x{:}t$ for adding the binding $x \mapsto t$ to the map. Duplicate declarations of the same variable in the same block are forbidden; with $x \notin \Delta$ we express that x is not bound in block Δ. We refer to a judgement $\Gamma \vdash e : t$, which reads "in context Γ, expression e has type t".

$$\frac{\Gamma. \vdash ss \Rightarrow \Gamma.\Delta}{\Gamma \vdash \{ss\} \Rightarrow \Gamma} \qquad \frac{\Gamma.\Delta \vdash e : t}{\Gamma.\Delta \vdash t\,x\,\texttt{=}\,e\texttt{;} \Rightarrow (\Gamma.\Delta, x{:}t)}\; x \notin \Delta$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e\texttt{;} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma. \vdash s_1 \Rightarrow \Gamma.\Delta_1 \qquad \Gamma. \vdash s_2 \Rightarrow \Gamma.\Delta_2}{\Gamma \vdash \texttt{if (}e\texttt{)}\, s_1\, \texttt{else}\, s_2 \Rightarrow \Gamma}$$

This judgement for statements is extended to sequences of statements $\Gamma \vdash ss \Rightarrow \Gamma'$ by the following rules (ε stands for the empty sequence):

$$\frac{}{\Gamma \vdash \varepsilon \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash s \Rightarrow \Gamma' \qquad \Gamma' \vdash ss \Rightarrow \Gamma''}{\Gamma \vdash s\; ss \Rightarrow \Gamma''}$$

2. Write syntax-directed *interpretation* rules for the *expressions* of Question 1. The form of the evaluation judgement should be $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$ where e denotes the expression to be evaluated in environment γ and the pair $\langle v; \gamma' \rangle$ denotes the resulting value and updated environment. You can assume a judgement $\gamma \vdash b \Downarrow v$ stating that block b evaluates to value v in environment γ.

   Alternatively, you can write the interpreter in pseudo code or Haskell (then assume a function `evalBlock` to be defined). A function `lookupVar` can be assumed if its behavior is described. In any case, the environment must be made explicit. (6p)

SOLUTION: The evaluation judgement $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$ for expressions is the least relation closed under the following rules.

$$\frac{}{\gamma \vdash \texttt{false} \Downarrow \langle 0; \gamma \rangle} \qquad \frac{}{\gamma \vdash \texttt{true} \Downarrow \langle 1; \gamma \rangle} \qquad \frac{}{\gamma \vdash i \Downarrow \langle i; \gamma \rangle}$$

$$\frac{i = \gamma(x)}{\gamma \vdash x\texttt{++} \Downarrow \langle i; \gamma[x = i{+}1] \rangle} \qquad \frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle i_2; \gamma'' \rangle}{\gamma \vdash e_1 \texttt{ + } e_2 \Downarrow \langle i_1{+}i_2; \gamma'' \rangle}$$

$$\frac{\gamma_0 \vdash e_1 \Downarrow \langle v_1; \gamma_1 \rangle \ldots \gamma_{n-1} \vdash e_n \Downarrow \langle v_n; \gamma_n \rangle \qquad x_1{=}v_1, \ldots, x_n{=}v_n \vdash b \Downarrow v}{\gamma_0 \vdash f\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} \Downarrow \langle v; \gamma_n \rangle}$$
with function definition $t\, f\, (t_1\, x_1, \ldots t_n\, x_n)\, b$

Herein, environment γ is a comma-separated list bindings of the form $x = v$. We write $\gamma[x = v]$ updating the value of x to v.

Question 5 (Compilation):

1. Write compilation schemes in pseudo code or Haskell for the statement, block, and expressions constructions of Question 1. The compiler should output symbolic JVM instructions (i.e. Jasmin assembler). It is not necessary to remember exactly the names of the instructions—only what arguments they take and how they work.

   Service functions like `addVar`, `lookupVar`, `lookupFun`, `newLabel`, `newBlock`, `popBlock`, and `emit` can be assumed if their behavior is described. (9p)

SOLUTION:

```
-- Compilation of expressions

compile (ETrue)  = emit (ldc 1)
compile (EFalse) = emit (ldc 0)
compile (EInt i) = emit (ldc i)

compile (ECall f es) = do
  m <- lookupFun f              -- get the Jasmin name of function f
  for (e ∈ es): compile e
  emit (invokestatic m)

compile (EPostIncr x) = do
  a <- lookupVar x
  emit (iload a)
  emit (dup)
  emit (ldc 1)
  emit (iadd)
  emit (store a)

compile (EPlus e e') = do
  compile e
  compile e'
  emit (iadd)

-- Compilation of statements

compile (SInit t x e) = do
  addVar t x              -- register local variable x, emit no code
  a <- lookupVar a
  compile e
  emit (istore a)

compile (SExp e) = do
  compile e
  emit (pop)
```

```
compile (SIfElse e s s') = do
  else, done <- newLabel
  compile e                         -- condition
  emit (ifeq else)                  -- if false, goto else:
  newBlock
  compile s                         -- then stm
  popBlock
  emit (goto done)                  -- jump over else stm
  emit (else:)
  newBlock
  compile s'                        -- else stm
  popBlock
  emit (done:)

compile (SBlock ss) = do
  newBlock
  for (s ∈ ss): compile s
  popBlock
```

2. Give the small-step semantics of the JVM instructions you used in the compilation schemes in part 1. Write the semantics in the form

$$i : (P, V, S) \longrightarrow (P', V', S')$$

where (P, V, S) is the program counter, variable store, and stack before execution of instruction i, and (P', V', S') are the respective values after the execution. For adjusting the program counter, you can assume that each instruction has size 1. (7p)

SOLUTION: Stack $S.v$ shall mean that the top value on the stack is v, the rest is S. Jump targets L are used as instruction addresses, and $P + 1$ is the instruction address following P.

instruction	state before		state after	
`goto` L	(P, V, S)	$\rightarrow$	(L, V, S)	
`ifeq` L	$(P, V, S.0)$	$\rightarrow$	(L, V, S)	
`ifeq` L	$(P, V, S.v)$	$\rightarrow$	$(P + 1, V, S)$	if $v \neq 0$
`iload` a	(P, V, S)	$\rightarrow$	$(P + 1, V, S.V(a))$	
`istore` a	$(P, V, S.v)$	$\rightarrow$	$(P + 1, V[a := v], S)$	
`ldc` i	(P, V, S)	$\rightarrow$	$(P + 1, V, S.i)$	
`iadd`	$(P, V, S.v.w)$	$\rightarrow$	$(P + 1, V, S.(v + w))$	
`dup`	$(P, V, S.v)$	$\rightarrow$	$(P + 1, V, S.v.v)$	
`pop`	$(P, V, S.v)$	$\rightarrow$	$(P + 1, V, S)$	
`invokestatic` m	$(P, V, S.v_1 \ldots v_n)$	$\rightarrow$	$(P + 1, V, S.v)$ where $v = m(v_1, \ldots, v_n)$	

Question 6 (Functional languages):

1. The following grammar describes a tiny simply-typed sub-language of Haskell.

x			identifier
n	$::=$	$0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \dots$	numeral
e	$::=$	$n \mid e + e \mid x \mid \lambda x \to e \mid e\, e$	expression
t	$::=$	$\mathsf{Int} \mid t \to t$	type

Application $e_1\, e_2$ is left-associative, the arrow $t_1 \to t_2$ is right-associative.

For the following typing judgements $\Gamma \vdash e : t$, decide whether they are valid or not. Your answer can be just "valid" or "not valid", but you may also provide a justification why some judgement is valid or invalid.

(a)		$\vdash \lambda x \to \lambda y \to (y\, x)\, 0$	$: \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$
(b)	$g : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$	$\vdash (g + 1)\,(\lambda x \to x)$	$: \mathsf{Int}$
(c)	$f : \mathsf{Int} \to \mathsf{Int}$	$\vdash \lambda x \to f\,(f\,(1 + (f\, x)))$	$: \mathsf{Int} \to \mathsf{Int}$
(d)	$x : \mathsf{Int} \to \mathsf{Int},\ g : \mathsf{Int}$	$\vdash x\,(g + 1)$	$: \mathsf{Int}$
(e)	$f : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$	$\vdash (\lambda x \to f\, x)\,(\lambda x \to f\,(\lambda x \to x)\, x)$	$: \mathsf{Int} \to \mathsf{Int}$

The usual rules for multiple-choice questions apply: For a correct answer you get 1 *point for a wrong answer* -1 *points. If you choose not to give an answer for a judgement, you get* 0 *points for that judgement. Your final score will be between* 0 *and* 5 *points, a negative sum is rounded up to* 0. (5p)

CLARIFICATION: There was a spurious parenthesis at the end of question (e): $(\lambda x \to f\, x)\,(\lambda x \to f\,(\lambda x \to x)\, x))$. Fixed. Announced (blackboard) during exam.

SOLUTION:

(a) not valid (y is not a function)
(b) not valid (g is a function, cannot add 1 to it)
(c) valid
(d) valid
(e) valid

2. Write a **call-by-name** interpreter for the functional language above, either with inference rules or in pseudo code or Haskell. (5p)

SOLUTION:

```haskell
type Var = String
data Exp
  = EInt Integer | EPlus Exp Exp
  | EVar Var | EAbs Var Exp | EApp Exp Exp

data Val = VInt Integer | VClos Var Exp Env
data Clos = Clos Exp Env
type Env = [(Var,Clos)]

eval :: Exp -> Env -> Maybe Val
eval e0 rho = case e0 of
  EInt n   -> return (VInt n)
  EAbs x e -> return (VClos x e rho)

  EPlus e f -> do
    VInt n <- eval e rho
    VInt m <- eval f rho
    return (VInt (n + m))

  EVar x   -> do
    Clos e rho' <- lookup x rho
    eval e rho'

  EApp f e -> do
    VClos x f' rho' <- eval f rho
    eval f' ((x, Clos e rho) : rho')
```