Programming Language Technology

Exam, 11 April 2017 at 8.30–12.30 in SB (Sven Hultins gata 6)

Course codes: Chalmers DAT151, GU DIT231. As re-exam, also DAT150, DIT229/230, and TIN321.
Exam supervision: Andreas Abel (+46 31 772 1731), visits at 9:30 and 11:30.

Grading scale: Max = 60p, VG = 5 = 48p, 4 = 36p, G = 3 = 24p.
Allowed aid: an English dictionary.
Exam review: Tuesday 25 April 2017 at 10-11 in room EDIT 8103.

Please answer the questions in English.

Question 1 (Grammars): Write a labelled BNF grammar that covers the following constructs of a C-like imperative language: A program is a list of statements. Types are `int` and `bool`. Statement constructs are:

- variable declarations (e.g. `int` x`;`), not multiple variables, no initial value
- expression statements (E`;`)
- `while` loops
- blocks: (possibly empty) lists of statements enclosed in braces

Expression constructs are:

- identifiers/variables
- integer literals
- pre-increments of identifiers (`++`x)
- greater-or-equal-than comparisons (E `>=` E')
- assignments of identifiers (x `=` E)

Greater-or-equal is non-associative and binds stronger than assignment. Parentheses around and expression are allowed and have the usual meaning. An example program would be:

```
int x; x = 0; while (10 >= ++x) {}
```

You can use the standard BNFC categories `Integer` and `Ident` as well as list short-hands, and `terminator`, `separator`, and `coercions` rules. (10p)

SOLUTION:

```
Program.  Prg  ::= [Stm]                         ;

SDecl.    Stm  ::= Type Ident ";"                ;
SExp.     Stm  ::= Exp ";"                       ;
SWhile.   Stm  ::= "while" "(" Exp ")" Stm       ;
SBlock.   Stm  ::= "{" [Stm] "}"                 ;

terminator Stm ""                                ;

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

EId.      Exp1 ::= Ident                         ;
EInt.     Exp1 ::= Integer                       ;
EPreIncr. Exp1 ::= "++" Ident                    ;
EGEq.     Exp  ::= Exp1 ">=" Exp1                ;
EAss.     Exp  ::= Ident "=" Exp                 ;

coercions Exp 1                                  ;
```

Question 2 (Lexing): A *string literal* is a character sequence of length ≥ 2 which starts and ends with double quotes `"`. Taking away both the starting and the ending `"`, we obtain a string in which `"` may only appear in the form `""`. Valid string literals are e.g.: `"Hi!"` or `"""Ol"`. Invalid string literals are e.g.: `B"` (does not start with double quotes) `"A` (does not end with double quotes), or `"""` (the middle part `"` is not valid since it is a single `"`).

To simplify matters, we represent character `"` by a and any other character by b. The valid string literals from above become $abbba$ and $aaabba$ and the invalid ones ba, ab, and aaa. Our alphabet thus becomes $\Sigma = \{a, b\}$.

1. Give a regular expression for string literals (using alphabet Σ). Demonstrate that your regular expression accepts the two valid examples and rejects the three invalid ones. (5p)

2. Give a deterministic or non-deterministic automaton for recognizing string literals (using alphabet Σ). Demonstrate that your automaton accepts the two valid examples and rejects the three invalid ones. (5p)

SOLUTION:

1. $r = a(b + aa)^*a$. For the proofs of acceptance, we use the compositional semantics of regular expressions. For the proofs of rejectance, we use derivatives. Other demonstrations are possible.

   (a) $b + aa$ accepts b, thus, $(b + aa)^*$ accepts bbb, thus $a(b + aa)^*a$ accepts $abbba$.

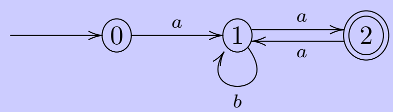
   (b) $b + aa$ accepts both b and aa, thus, $(b + aa)^*$ accepts $aabb$, thus, $a(b + aa)^*a$ accepts $aaabba$.

   (c) $r/ba = a(b + aa)^*a/ba = \emptyset$ which does not contain the empty word.

   (d) $r/ab = a(b + aa)^*a/ab = (b + aa)^*a/b = (b + aa)^*a$ which does not contain the empty word.

   (e) $r/aaa = a(b + aa)^*a/aaa = (b + aa)^*a/aa = (b + aa)^*a$ which does not contain the empty word.

2. A possible non-deterministic automaton uses three states $S = \{0, 1, 2\}$ with start state 0 and accepting state 2 and the following transitions.



   (This automaton could easily be made deterministic by adding an error state, reachable from 0 and 2 by character b.) To demonstrate acceptance or rejectance, we simply run the automaton on the input. We denote a run by the sequence of states the automaton goes through.

   (a) $abbba$ is accepted by run 011112.

   (b) $aaabba$ is accepted by run 0121112.

   (c) ba is stuck in state 0.

   (d) ab leads to run 011 ending in a non-accepting state.

   (e) aaa leads to run 0121 ending in a non-accepting state.

Question 3 (Parsing): Consider the following BNF-Grammar for boolean expressions (written in `bnfc`). The starting non-terminal is `D`.

```
Or.     D ::= D "|" C   ;  -- Disjunctions
Conj.   D ::= C         ;

And.    C ::= C "&" A   ;  -- Conjunctions
Atom.   C ::= A         ;

TT.     A ::= "true"    ;  -- Atoms
FF.     A ::= "false"   ;
Var.    A ::= "x"       ;
Parens. A ::= "(" D ")" ;
```

Step by step, trace the LR-parsing of the expression

```
false | x & true
```

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

SOLUTION: The actions are `S` (shift), `R` (reduce with rule(s)), and `Accept`.

```
Stack     . Input              // Action(s)          (rules)
-----------------------------------------------------------------
          . false | x & true   //  SR: "false" -> A  (FF)
A         . | x & true         //   R: A -> C -> D   (Atom, Conj)
D         . | x & true         // SSR: "x" -> A      (Var)
D | A     . & true             //   R: A -> C        (Atom)
D | C     . & true             // SSR: "true" -> A   (TT)
D | C & A                      //   R: C & A -> C    (And)
D | C                          //   R: D | C -> D    (Or)
D                              // Accept
```

Question 4 (Type checking and evaluation):

1. Write syntax-directed *type checking* rules for the *statement* forms and lists of Question 1. The typing environment must be made explicit. You can assume a type-checking judgement for expressions.

   Alternatively, you can write the type-checker in pseudo code or Haskell.

   Please pay attention to scoping details; in particular, the program

   ```
   while (0 >= 0) int x; x = 0;
   ```

   should not pass your type checker! (5p)

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 use a judgement $\Gamma \vdash e : t$, which reads "in context Γ, expression e has type t".

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

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma. \vdash s \Rightarrow \Gamma.\Delta}{\Gamma \vdash \texttt{SWhile}\, e\, s \Rightarrow \Gamma} \qquad \frac{\Gamma. \vdash ss \Rightarrow \Gamma.\Delta}{\Gamma \vdash \texttt{SBlock}\, ss \Rightarrow \Gamma}$$

This judgement is extended to sequences of statements $\Gamma \vdash ss \Rightarrow \Gamma'$ by the following rules:

$$\frac{}{\Gamma \vdash \texttt{SNil} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash s \Rightarrow \Gamma' \qquad \Gamma' \vdash ss \Rightarrow \Gamma''}{\Gamma \vdash \texttt{SCons}\, s\, ss \Rightarrow \Gamma''}$$

Alternative solution: Lists of statements are denoted by ss and ε is the empty list. The judgement $\Gamma \vdash ss$ reads "in context Γ, the sequence of statements ss is well-formed". Here, concrete syntax is used for the statements:

$$\frac{}{\Gamma \vdash \varepsilon} \qquad \frac{\Gamma.\Delta \vdash e : t \qquad \Gamma.\Delta, x : t \vdash ss}{\Gamma.\Delta \vdash t\, x; ss}\ x \notin \Delta \qquad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash ss}{\Gamma \vdash e; ss}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma. \vdash s \qquad \Gamma \vdash ss}{\Gamma \vdash \texttt{while}(e)s\ ss} \qquad \frac{\Gamma. \vdash ss \qquad \Gamma \vdash ss'}{\Gamma \vdash \{ss\}ss'}$$

Possible Haskell solution:

```
chkStm :: Stm -> StateT [Map Ident Type] Maybe ()
chkStm (SExp e)     = do
  chkExp e Nothing                       -- Check e is well-typed
chkStm (SDecl t x)  = do
  (delta : gamma) <- get                 -- Get context
  guard $ Map.notMember x delta          -- No duplicate binding!
  put $ Map.insert x t delta : gamma -- Add binding
chkStm (SWhile e s) = do
  chkExp e (Just TBool)                  -- Check e against bool
  modify (Map.empty :)                   -- Push new block
  chkStm s
  modify tail                            -- Pop top block
chkStm (SBlock ss) = do
  modify (Map.empty :)                   -- Push new block
  mapM_ chkStm ss
  modify tail                            -- Pop top block
```

2. Write syntax-directed *interpretation* rules for the *expression* forms of Question 1. The environment must be made explicit, as well as all possible side effects.

   Alternatively, you maybe write an interpeter in pseudo code or Haskell. (5p)

SOLUTION:

The judgement $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$ reads "in environment γ, evaluation of the expression e results in value v and environment γ'".

$$\frac{}{\gamma \vdash \texttt{EInt}\, i \Downarrow \langle i; \gamma \rangle} \qquad \frac{}{\gamma \vdash \texttt{EVar}\, x \Downarrow \langle \gamma(x); \gamma \rangle}$$

$$\frac{}{\gamma \vdash \texttt{EPreIncr}\, x \Downarrow \langle \gamma(x) + 1; \gamma[x := \gamma(x) + 1] \rangle}$$

$$\frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma_1 \rangle \qquad \gamma_1 \vdash e_2 \Downarrow \langle i_2; \gamma_2 \rangle}{\gamma \vdash \texttt{EGEq}\, e_1\, e_2 \Downarrow \langle i_1 \geq i_2; \gamma_2 \rangle} \qquad \frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash \texttt{EAss}\, x\, e \Downarrow \langle v; \gamma'[x := v] \rangle}$$

Question 5 (Compilation):

1. Write compilation schemes in pseudo code for each of the *expression* constructions in Question 1 generating JVM (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. (6p)

SOLUTION:

```
compile (EVar x) = do
  a <- lookupVar x
  emit (iload a)           -- load value of x onto stack

compile (EInt i) = do
  emit (ldc i)             -- put i onto stack

compile (EAss x e) = do
  compile e                -- value of e is on stack
  a <- lookupVar x
  istore a                 -- store value
  iload a                  -- put value back on stack

compile (EPreIncr x) = do
  a <- lookupVar x
  emit (iload a)           -- load value of x onto stack
  emit (ldc 1)             -- increment
  emit (iadd)
  emit (istore a)          -- store value
  emit (iload a)           -- put value back on stack

compile (EGEq e1 e2) = do
  LDone  <- newLabel
  emit (ldc 1)             -- push "true"
  compile e1
  compile e2
  emit (if_icmpge LDone)   -- if greater or equal, then done
  emit (pop)               -- remove "true"
  emit (ldc 0)             -- push "false"
  emit (LDone:)
```

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) are 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. (6p)

SOLUTION:

$$\begin{array}{llllll}
\texttt{ldc}\ a & : & (P, V, S) & \longrightarrow & (P+1, V, \quad S.a) & \\
\texttt{iload}\ x & : & (P, V, S) & \longrightarrow & (P+1, V, \quad S.V(x)) & \\
\texttt{istore}\ x & : & (P, V, S.a) & \longrightarrow & (P+1, V[x=a], S) & \\
\texttt{pop} & : & (P, V, S.a) & \longrightarrow & (P+1, V, \quad S) & \\
\texttt{iadd} & : & (P, V, S.a.b) & \longrightarrow & (P+1, V, \quad S.(a+b)) & \\
\texttt{if_icmpge}\ L & : & (P, V, S.a.b) & \longrightarrow & (L, \quad V, \quad S) & \text{if } a \geq b \\
\texttt{if_icmpge}\ L & : & (P, V, S.a.b) & \longrightarrow & (P+1, V, \quad S) & \text{otherwise}
\end{array}$$

Question 6 (Functional languages):

1. For lambda-calculus expressions we use the grammar

$$e ::= n \mid x \mid \lambda x \to e \mid e\, e$$

and for simple types $t ::= \texttt{int} \mid t \to t$. Non-terminal x ranges over variable names and n over integer constants 0, 1, etc.

For the following typing judgements $\Gamma \vdash e : t$, decide whether they are valid or not. Your answer should be just "valid" or "not valid".

(a) $\vdash \lambda x \to \lambda y \to (f\, x)\, y : \mathsf{int} \to (\mathsf{int} \to \mathsf{int})$.

(b) $y : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \vdash y\, (\lambda x \to 1) : \mathsf{int}$.

(c) $f : \mathsf{int} \to \mathsf{int} \vdash \lambda x \to f\, (f\, x) : \mathsf{int} \to \mathsf{int}$.

(d) $y : \mathsf{int} \to \mathsf{int}, f : \mathsf{int} \vdash f\, y : \mathsf{int}$.

(e) $f : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \vdash (\lambda x \to f\, (x\, x))\, (\lambda \to f\, (x\, x)) : \mathsf{int} \to \mathsf{int}$.

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)

SOLUTION:

(a) not valid (f is unbound)

(b) valid

(c) valid

(d) not valid (f does not have a function type)

(e) not valid (self application $x\,x$ is not typable)

2. Write a call-by-value interpreter for above lambda-calculus either with inference rules, or in pseudo-code or Haskell. (5p)

SOLUTION:

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

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

eval :: Exp -> Env -> Maybe Val
eval e0 rho = case e0 of
  EInt n   -> return $ VInt n
  EAbs x e -> return $ VClos x e rho
  EVar x   -> lookup x rho
  EApp f e -> do
    VClos x e' rho' <- eval f rho
    v               <- eval e rho
    eval e' $ (x,v):rho'
```