Programming Language Technology

Exam, 13 January 2022 at 08.30 – 12.30 in HA1-4

Course codes: Chalmers DAT151, GU DIT231.
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 2022 13.30-15.00 in EDIT meeting room 6128 (6th 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 `}`
- Statements:
  - block
  - variable declaration, e.g., `int x;`
  - statement formed from an expression by adding a semicolon `;`
  - `while` statement
- Expressions, from highest to lowest precedence:
  - parenthesized expression, identifier, integer literal
  - addition (+), left associative
  - less-than comparison (<), non-associative
  - assignment (`x = e`), right associative
- Type: `int` or `bool`

You can use the standard BNFC categories `Integer` and `Ident` but *none* of the BNFC pragmas (`coercions`, `terminator`, `separator` ...). An example program is:

```
int main () {
  int x;  x = 0;
  while ((x = x + 1) < 10) int x;
}
```

(10p)

SOLUTION:

```
Program.  Prg   ::= "int" "main" "(" ")" "{" Stms "}"    ;

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

SNil.     Stms  ::=                                      ;
SCons.    Stms  ::= Stm Stms                             ;

EId.      Exp3  ::= Ident                                ;
EInt.     Exp3  ::= Integer                              ;
EPlus.    Exp2  ::= Exp2 "+" Exp3                        ;
ELt.      Exp1  ::= Exp2 "<" Exp2                        ;
EAss.     Exp   ::= Ident "=" Exp                        ;

_.        Exp3  ::= "(" Exp ")"                          ;
_.        Exp2  ::= Exp3                                 ;
_.        Exp1  ::= Exp2                                 ;
_.        Exp   ::= Exp1                                 ;

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

Question 2 (Lexing): An *identifier* be a non-empty sequence of letters, digits and underscores, with the following limitation: *There must be at least one letter between any two of these positions: beginning, end, and any underscore position.* In other terms, if you cut the identifier into words at each underscore, each of the words needs to contain at least one letter.

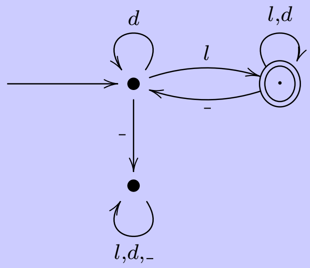
Letters be subsumed under the non-terminal l and *digits* under d, thus, the alphabet is just $\{l, d, \_\}$.

1. Give a regular expression for identifiers.
2. Give a deterministic finite automaton for identifiers with no more than 9 states.

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

SOLUTION:

1. RE: $d^*l(l+d)^*(\_d^*l(l+d)^*)^*$ OR $(d^*l)^+d^*(\_(d^*l)^+d^*)^*$ OR ...
2. DFA:



Question 3 (LR Parsing): Consider the following labeled BNF-Grammar. The starting non-terminal is `C`.

```
Cond.   C ::= S "<" S ;
Sum.    S ::= S "+" X ;
Atom.   S ::= X       ;

A.      X ::= "a"     ;
B.      X ::= "b"     ;
C.      X ::= "c"     ;
D.      X ::= "d"     ;
```

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

```
a + b < c + d
```

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.

```
          . a + b < c + d    -- shift
a         . + b < c + d      -- reduce with rule A
X         . + b < c + d      -- reduce with rule Atom
S         . + b < c + d      -- shift 2
S + b     . < c + d          -- reduce with rule B
S + X     . < c + d          -- reduce with rule Sum
S         . < c + d          -- shift 2
S < c     . + d              -- reduce with rule C
S < X     . + d              -- reduce with rule Atom
S < S     . + d              -- shift 2
S < S + d .                  -- reduce with rule D
S < S + X .                  -- reduce with rule Sum
S < S     .                  -- reduce with rule Cond
C         .                  -- 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".

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

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e\texttt{;} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma. \vdash s \Rightarrow \Gamma.\Delta}{\Gamma \vdash \texttt{while (}e\texttt{)}\; s \Rightarrow \Gamma}$$

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.

   Alternatively, you can write the interpreter in pseudo code or Haskell. 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.

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

$$\frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle i_2; \gamma'' \rangle}{\gamma \vdash e_1 \texttt{ + } e_2 \Downarrow \langle i_1{+}i_2; \gamma'' \rangle} \qquad \frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle i_2; \gamma'' \rangle}{\gamma \vdash e_1 \texttt{ < } e_2 \Downarrow \langle i_1{<}i_2; \gamma'' \rangle}$$

$$\frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash x\texttt{=}e \Downarrow \langle v; \gamma'[x = v] \rangle}$$

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: The state of the compiler has the following components:

(a) A potentially infinite supply of unique label names. Service function `newLabel` takes out one label from this supply and returns it.

(b) A stack of blocks each of which maps variable identifiers to JVM local variable addresses. Procedure `newBlock` pushes a new empty block onto the stack, `popBlock` removes the top block from the stack. Function `addVar` x binds x to the least yet-unallocated address in the top block and returns this address. Function `lookupVar` x searches, starting at the top of the stack, the blocks for the address bound to x and returns this address.

(c) A list of JVM instructions generated by the compiler so far. Function `emit`(i) appends instruction i to this list.

Compilation of expressions, statements and blocks is performed in accordance with the Haskell-like pseudo-code below:

```
-- We annotate the stack usage by comments of the form S<nnn>.
-- This is relative to the stack usage upon entry, which we set to 0.

-- Compilation of expressions (increase stack usage by 1)

compile (EInt i) = do      -- S0
  emit (ldc i)             -- S1

compile (EId x) = do       -- S0
  a <- lookupVar x         -- variable x has address a in store
  emit (iload a)           -- S1

compile (EAss x e) = do    -- S0
  compile e                -- S1
  a <- lookupVar x
  emit (istore a)          -- S0
  emit (iload a)           -- S1

compile (EPlus e e') = do  -- S0
  compile e                -- S1
  compile e'               -- S2
  emit (iadd)              -- S1
```

```
compile (ELt e e') = do    -- S0
  done <- newLabel
  emit (ldc 1)             -- S1; speculate that e < e' holds
  compile e                -- S2
  compile e'               -- S3
  emit (if_icmplt done)    -- S1; test e < e'
  emit (pop)               -- S0; test failed, replace 1 by 0
  emit (ldc 0)             -- S1
  emit (done:)             -- S1

-- Compilation of statements (preserve stack usage)

compile (SDecl t x) = do   -- S0
  addVar x                 -- register local variable x, emit no code

compile (SExp e) = do      -- S0
  compile e                -- S1
  emit (pop)               -- S0

compile (SWhile e s) = do  -- S0
  start, done <- newLabel
  emit (start:)            -- S0
  compile e                -- S1;  condition
  emit (ifeq done)         -- S0; if false, exit loop
  newBlock
  compile s                -- S0
  popBlock
  emit (goto start)        -- S0; rerun loop
  emit (done:)             -- S0

-- Compilation of blocks

compile (SBlock ss) = do   -- S0
  newBlock
  for (s : ss)
    compile s              -- S0
  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 $v \neq 0$
`if_icmplt` L	$(P, V, S.v.w)$	$\rightarrow$	(L, V, S)	if $v < w$
`if_icmplt` L	$(P, V, S.v.w)$	$\rightarrow$	$(P + 1, V, S)$	unless $v < w$
`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`	$(P, V, S.v.w)$	$\rightarrow$	$(P + 1, V, S.(v + w))$	
`pop`	$(P, V, S.v)$	$\rightarrow$	$(P + 1, V, S)$	

Question 6 (Functional languages):

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

x			identifier
i	$::=$	$0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots$	integer literal
e	$::=$	$i \mid e + e \mid x \mid \lambda x \to e \mid e\, e$	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)	$x : \mathsf{Int}$	$\vdash \lambda y \to (y\, x)\, 0$	$: (\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
(b)	$g : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$	$\vdash g\, (\lambda x \to g\, x)$	$: \mathsf{Int}$
(c)	$f : \mathsf{Int} \to \mathsf{Int}$	$\vdash \lambda x \to f\, (f\, 1 + f\, x)$	$: \mathsf{Int} \to \mathsf{Int}$
(d)	$x : \mathsf{Int},\ g : \mathsf{Int} \to \mathsf{Int}$	$\vdash x\, (g + 1)$	$: \mathsf{Int}$
(e)	$f : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$	$\vdash (\lambda x \to f\, x)\, (\lambda f \to \lambda x \to f\, x)$	$: \mathsf{Int} \to \mathsf{Int}$

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) valid
(b) not valid (g is self-applied)
(c) valid
(d) not valid (g is a function, cannot add 1 to it)
(e) not valid (f has order 2, cannot be applied to order 2 argument)

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

SOLUTION:

```haskell
type Var = String
data Exp = EVar Var | EAbs Var Exp | EApp Exp Exp
         | EInt Integer | EPlus 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
  EAbs x e -> return (VClos x e rho)
  EVar x -> lookup x rho
  EInt n -> return (VInt n)

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

  EPlus e1 e2 -> do
    VInt i1 <- eval e1 rho
    VInt i2 <- eval e2 rho
    return (VInt (i1 + i2))
```