Programming Language Technology

Exam, 08 January 2018 at 8.30–12.30 in J

Course codes: Chalmers DAT150/151, GU DIT231. As re-exam, also TIN321 and DIT229/230.
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 16 January 2017 at 10.30-12 in room EDIT 6128.

Please answer the questions in English. Questions requiring answers in code can be answered in any of: C, C++, Haskell, Java, or precise pseudocode.

For any of the six questions, an answer of roughly one page should be enough.

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

- statements:
  - blocks: lists of statements (possibly empty) in curly brackets { }
  - variable initialization statement: a type followed by an identifier, the equals sign, an initializing expression, and a semicolon, e.g. `int x = 4;`
  - return statements: an expression between keyword `return` and a semicolon, e.g. `return 1;`
- types: `int`
- expressions:
  - integer literals
  - subtraction `-`
  - multiplication `*`
  - pre-increment of variables `++x`
  - function calls `x(e,...,e)` with zero or more arguments
  - parenthesized expressions `(e)`.

  Both arithmetic operations are left associative. Multiplication binds stronger than subtraction.

An example statement is:

```
{ int x = f (0, 1); return ++x - (2 - 3) - 4 * g();
  int y = 5;        return ++y; }
```

You can use the standard BNFC categories `Integer` and `Ident`. **Do not** use list categories or `terminator`/`separator` rules. **Do not** use `coercions` either. (10p)

SOLUTION:

```
SRet.     Stm  ::= "return" Exp ";"       ;
SBlock.   Stm  ::= "{" Stms "}"           ;
SInit.    Stm  ::= Type Ident "=" Exp ";" ;

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

TInt.     Type ::= "int"                  ;

EInt.     Exp2 ::= Integer                ;
ECall.    Exp2 ::= Ident "(" Exps ")"     ;
EPreIncr. Exp2 ::= "++" Ident             ;
ETimes.   Exp1 ::= Exp1 "*" Exp2          ;
EMinus.   Exp  ::= Exp  "-" Exp1          ;

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

ENil.     Exps ::=                        ;
ESg.      Exps ::= Exp                    ;
ECons.    Exps ::= Exp "," Exps           ;
```

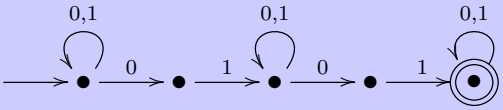
Question 2 (Lexing): Consider the alphabet $\Sigma = \{0, 1\}$ and the language L of bit-streams that contains the sequence 01 *at least twice*.

1. Give a regular expression for language L.
2. Give a non-deterministic finite automation for L.
3. Give a deterministic finite automaton for L.

(8p)

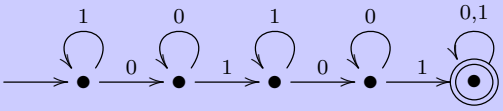
SOLUTION:

1. RE: $(0 + 1)^*01(0 + 1)^*01(0 + 1)^*$
2. NFA:



   Of course, the following DFA would also be a possible solution for the NFA. (Every DFA is trivially a NFA.)

3. DFA:



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

```
S1.     S ::= S ";" C   ;
S2.     S ::= C         ;

C1.     C ::= C "," A   ;
C2.     C ::= A         ;

AA.     A ::= "alice"   ;
AB.     A ::= "bob"     ;
AC.     A ::= "chris"   ;
```

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

```
alice ; bob , chris
```

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 and input are separated by a dot.

```
a = alice
b = bob
c = chris

          . a ; b , c   -- S        AA      C2      S2
a         .   ; b , c   -- RRR:  a ----> A ----> C ----> S
S         .   ; b , c   -- SS       AB      C2
S ; b     .       , c   -- RR:   b ----> A ----> C
S ; C     .       , c   -- SS       AC
S ; C , c .             -- R:    c ----> A
S ; C , A .             -- R with C1
S ; C     .             -- R with S1
S         .             -- Accept
```

Question 4 (Type checking and evaluation):

1. Write syntax-directed *type checking* rules for the *expression* forms of Question 1. The typing environment must be made explicit.

   Alternatively, you can write the type-checker in pseudo code or Haskell. (6p)

SOLUTION: We use a judgement $\Gamma \vdash e : t$ that expresses that expression e has type t in context Γ.

A context Γ is a finite map from variables to types. The global signature Σ maps functions to their function types.

$$\frac{i \in \texttt{Integer}}{\Gamma \vdash \texttt{EInt}\, i : \texttt{TInt}} \qquad \frac{\Gamma \vdash e_1 : \texttt{TInt} \qquad \Gamma \vdash e_2 : \texttt{TInt}}{\Gamma \vdash \texttt{EMinus}\, e_1\, e_2 : \texttt{TInt}}$$

$$\frac{\Gamma(x) = \texttt{TInt}}{\Gamma \vdash \texttt{EPreIncr}\, x : \texttt{TInt}} \qquad \frac{\Gamma \vdash e_1 : \texttt{TInt} \qquad \Gamma \vdash e_2 : \texttt{TInt}}{\Gamma \vdash \texttt{ETimes}\, e_1\, e_2 : \texttt{TInt}}$$

$$\frac{\Sigma(f) = (t_1, \ldots, t_n) \to t \qquad \Gamma \vdash e_i : t_i \text{ for } i{=}1..n}{\Gamma \vdash \texttt{ECall}\, f\, e_{1..n} : t}$$

2. Write syntax-directed *interpretation* rules for the *statement* forms and lists of Question 1. The environment must be made explicit, as well as all possible side effects. You can assume an interpreter for expressions.

   Alternatively, you can write the interpreter in pseudo code or Haskell. (6p)

SOLUTION: Environments γ are stacks of blocks δ, each of which is a map from variables x to values v, which are integers. By $\gamma, x := v$ we denote the environment γ' that results from adding the binding of x to v to the top-most block of γ.

The judgement $\gamma \vdash s \Downarrow \langle r; \gamma' \rangle$ reads "in environment γ, evaluation of the statement s results in r and environment γ'". A result is either return value v or the special symbol ε, meaning "continue".

We use an auxiliary judgement $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$ that states that "in environment γ, expression e evaluates to v, with side-effects producing a new environment γ'".

$$\frac{\gamma. \vdash ss \Downarrow \langle r; \gamma'.\delta' \rangle}{\gamma \vdash \texttt{SBlock}\, ss \Downarrow \langle r; \gamma' \rangle} \qquad \frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash \texttt{SInit}\, t\, x\, e \Downarrow \langle \varepsilon; \gamma', x := v \rangle}$$

$$\frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash \texttt{SRet}\, e \Downarrow \langle v; \gamma' \rangle} \qquad \frac{}{\gamma \vdash \texttt{SNil} \Downarrow \langle \varepsilon; \gamma \rangle}$$

$$\frac{\gamma \vdash s \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash \texttt{SCons}\, s\, ss \Downarrow \langle v; \gamma' \rangle} \qquad \frac{\gamma \vdash s \Downarrow \langle \varepsilon; \gamma' \rangle \qquad \gamma' \vdash ss \Downarrow \langle r; \gamma'' \rangle}{\gamma \vdash \texttt{SCons}\, s\, ss \Downarrow \langle r; \gamma'' \rangle}$$

Question 5 (Compilation):

1. Write compilation schemes in pseudo-code for each of the grammar constructions in 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. (8p)

SOLUTION:

```
compile (EInt i):
  ldc i

compile (EMinus e1 e2):
  compile e1
  compile e1
  isub

compile (ETimes e1 e2):
  compile e1
  compile e2
  imul

compile (EPreIncr x):
  a <- lookupvar x
  iload a
  ldc 1
  iadd
  dup
  istore a

compile (ECall f es):
  for (e : es)
    compile e
  invokestatic f

compile (SBlock ss):
  newBlock
  for (s : ss)
    compile s
  popBlock

compile (SInit t x e):
  compile e
  a <- newVar x
  istore a

compile (SRet e):
  compile e
  ireturn
```

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.

You can omit instructions for function call and return. (4p)

SOLUTION:

`ldc` a	:	(P, V, S)	$\longrightarrow (P+1, V,$	$S.a)$
`iload` x	:	(P, V, S)	$\longrightarrow (P+1, V,$	$S.V(x))$
`istore` x	:	$(P, V, S.a)$	$\longrightarrow (P+1, V[x=a],$	$S)$
`isub`	:	$(P, V, S.a.b)$	$\longrightarrow (P+1, V,$	$S.(a-b))$
`imul`	:	$(P, V, S.a.b)$	$\longrightarrow (P+1, V,$	$S.(a*b))$
`invokestatic` f	:	$(P, V, a_{0..n-1})$	$\longrightarrow (f,\quad (i \mapsto a_i),$	$\varepsilon)$
`ireturn`	:	$(P, V, S.a)$	$\longrightarrow ?$	

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) $\vdash \lambda x \to \lambda y \to (f\, x)\, 0 : \mathsf{int} \to (\mathsf{int} \to \mathsf{int})$.

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

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

(d) $x : \mathsf{int} \to \mathsf{int}, g : \mathsf{int} \vdash g\, x : \mathsf{int}$.

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

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 (g 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'
```