Programming Language Technology

Exam, 25 August 2022 at 14.00 – 18.00 in SB Multisal

Course codes: Chalmers DAT151, GU DIT231.
Exam supervision: Andreas Abel (+46 31 772 1731), visits at 15:30.

Grading scale: Max = 60p, VG = 5 = 48p, 4 = 36p, G = 3 = 24p.
Allowed aid: an English dictionary.
Exam review: Contact examiner Andreas Abel for a review (office EDIT 6111).

Please answer the questions in English.

Question 1 (Grammars): Write a labelled BNF grammar that covers the following kinds of constructs of C/C++ (sublanguage of lab 2):

- Program: a sequence of unary function definitions.
- Function definition: type, identifier, exactly one parameter in parentheses, block.
- Parameter: type followed by identifier, e.g. `int x`.
- Block: a sequence of statements enclosed between `{` and `}`
- Statements:
  - block
  - single variable declaration, e.g., `int x;`
  - expression followed by semicolon
  - return statement
  - while statement
- Expressions, from highest to lowest precedence:
  - parenthesized expression, identifier, integer literal, unary function call
  - addition (+), left associative
  - less-than comparison (<), non-associative
  - assignment, right associative
- Type: `bool` or `int` or `void`

You can use the standard BNFC categories `Integer` and `Ident` and the list pragmas `terminator` and `separator`, but *not* the `coercions` pragma. An example program is:

```
int f (int z) {
  int x;
  x = (z = 7) + z;
  while (z < x) {
    int x;
    printInt(x = z + 2);
    z = x;
  }
  return x;
}
```

(10p)

SOLUTION:

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

DFun.      Def   ::= Type Ident "(" Type Ident ")" "{" [Stm] "}"  ;
terminator Def ""                                                 ;

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

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

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

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

Question 2 (Lexing): An *non-nested PASCAL/ML comment* starts with `(*` and ends with `*)` and can have any characters in between (but not the comment-end sequence `*)` of course). Also, `(*)` is *not* a valid comment.

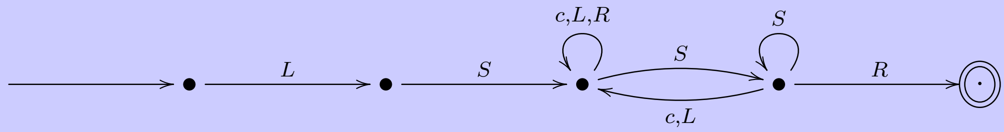
1. Give a deterministic finite automaton for such comments with no more than 8 states. Remember to mark initial and final states appropriately.

2. Give a regular expression for such comments.

Work in the alphabet $\{L, R, S, c\}$ distinguishing 4 tokens: L for `'('`, R for `')'`, S for `'*'`, and c where c stands for *any other character*. (6p)

SOLUTION:

1. DFA:



2. RE: E.g. $LS\Big(\big(c \mid L \mid R\big)^* \big(S^+(c \mid L)\big)^*\Big)^* S^+R$

   or $LS\Big(c \mid L \mid R \mid \big(S^+(c \mid L)\big)\Big)^* S^+R$

Question 3 (LR Parsing): Use your grammar from Question 1. Step by step, trace the shift-reduce parsing of the expression `x = z + 2` 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 = z + 2  -- shift
Ident                   . = z + 2    -- shift 2x
Ident = Ident           . + 2        -- reduce with rule EId
Ident = Exp2            . + 2        -- reduce with coercion rule
Ident = Exp1            . + 2        -- shift 2x
Ident = Exp1 + Integer  .            -- reduce with rule EInt
Ident = Exp1 + Exp2     .            -- reduce with rule EPlus
Ident = Exp1            .            -- reduce with coercion rule
Ident = Exp             .            -- reduce with rule EAss
Exp                     .            -- 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 $\Gamma \vdash_t s \Rightarrow \Gamma'$ where s is a statement or list of statements, t the return type, 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 and the return type must be made explicit. (6p)

SOLUTION: 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".

$$\frac{\Gamma. \vdash_t ss \Rightarrow \Gamma.\Delta}{\Gamma \vdash_t \{ss\} \Rightarrow \Gamma} \qquad \frac{}{\Gamma.\Delta \vdash_t t'\,x\texttt{;} \Rightarrow (\Gamma.\Delta, x{:}t')}\ x \notin \Delta$$

$$\frac{\Gamma \vdash e : t'}{\Gamma \vdash_t e\texttt{;} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash_t \texttt{return}\,e\texttt{;} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma. \vdash_t s \Rightarrow \Gamma.\Delta}{\Gamma \vdash_t \texttt{while (}e\texttt{)}\ s \Rightarrow \Gamma}$$

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

$$\frac{}{\Gamma \vdash_t \varepsilon \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash_t s \Rightarrow \Gamma' \qquad \Gamma' \vdash_t ss \Rightarrow \Gamma''}{\Gamma \vdash_t s\,ss \Rightarrow \Gamma''}$$

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 $\gamma \vdash ss \Downarrow v$ that yields the return value v when executing the block ss of a function.

   Alternatively, you can write the interpreter in pseudo code or Haskell (then assume **`evalBlock`**). 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.

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

$$\frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle \qquad x{=}v \vdash ss \Downarrow v'}{\gamma \vdash f(e) \Downarrow \langle v'; \gamma \rangle} \text{ function } t\ f\ (t'\ x)\{ss\} \text{ defined}$$

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

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

Herein, environment γ is map from identfiers to integers. Boolean true is represented by integer 1, and false by 0.

Question 5 (Compilation):

1. Translate the example program of Question 1 to Jasmin. It is not necessary to remember exactly the names of the JVM instructions—only what arguments they take and how they work. Make clear which instructions come from which statement, and determine the stack and local variable limits. (8p)

SOLUTION:

```
;; int f (int z) {...}

.method public static f(I)I
.limit locals 3
.limit stack  2

;; int x;
;; x = (z = 7) + z;

bipush 7
istore_0
iload_0
iload_0
iadd
istore_1

;; while (z < x)

L0:
iload_0
iload_1
if_icmpge L1

;; int x;
;; printInt (x = (z + 2));

iload_0
iconst_2
iadd
istore_2
iload_2
invokestatic Runtime/printInt(I)V

;; z = x;

iload_2
istore_0
goto L0
L1:
```

```
;; return x;

iload_1
ireturn

.end method
```

2. Give the small-step semantics of the JVM instructions you used in the Jasmin code in part 1 (except for return instructions). 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. (6p)

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)	$\to$	(L, V, S)	
`if_icmpge` L	$(P, V, S.v.w)$	$\to$	(L, V, S)	if $v \geq w$
`if_icmpge` L	$(P, V, S.v.w)$	$\to$	$(P+1, V, S)$	unless $v \geq w$
`iload` a	(P, V, S)	$\to$	$(P+1, V, S.V(a))$	
`istore` a	$(P, V, S.v)$	$\to$	$(P+1, V[a := v], S)$	
`iconst` i	(P, V, S)	$\to$	$(P+1, V, S.i)$	
`bipush` i	(P, V, S)	$\to$	$(P+1, V, S.i)$	
`iadd`	$(P, V, S.v.w)$	$\to$	$(P+1, V, S.(v+w))$	
`invokestatic` m	$(P, V, S.v_1 \ldots v_n)$	$\to$	$(P+1, V, S.v)$ 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
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. Application binds strongest, then addition, then λ-abstraction.

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)

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')
```