Programming Language Technology

Exam, 28 August 2025 at 14.00 – 18.00 in SB3 Multisal

Course codes: Chalmers DAT151, GU DIT231.
Exam supervision: Andreas Abel (+46 31 772 1731), visits at 15:00 and 17:00.

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 function definitions.
- Function definition: type, identifier, comma-separated parameter list in parentheses, block.
- Parameter: type followed by identifier, e.g. `int x`.
- Block: a sequence of statements enclosed between `{` and `}`
- Statements:
  - block
  - initializing variable declaration, e.g., `int x = 5;`
  - return statement
  - if-else statement
- Expressions, from highest to lowest precedence:
  - parenthesized expression, integer literal (e.g. `42`), identifier (e.g. `x`)
  - less-or-equal-than comparison (`<=`), non-associative
  - short-circuiting disjunction (`||`), left associative
  - assignment to identifiers (`x = e`), right associative
- Type: `int` or `bool`

You can use the standard BNFC categories `Integer` and `Ident`, and any of the BNFC pragmas (`coercions`, `terminator`, `separator` ...). An example program is:

```
bool f (int x, bool b) {
  int y = 127;
  if (b || (y = 42) <= x) {
    bool x = (y <= 100);
    return (x || b);
  } else return (x <= 55);
}
```

(10p)

SOLUTION:

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

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

ADecl.     Arg   ::= Type Ident                             ;
separator  Arg ","                                          ;

SBlock.    Stm   ::= "{" [Stm] "}"                          ;
SInit.     Stm   ::= Type Ident "=" Exp ";"                 ;
SReturn.   Stm   ::= "return" Exp ";"                       ;
SIfElse.   Stm   ::= "if" "(" Exp ")" Stm "else" Stm        ;
terminator Stm ""                                           ;

EId.       Exp3  ::= Ident                                  ;
EInt.      Exp3  ::= Integer                                ;
ELEq.      Exp2  ::= Exp3 "<="  Exp3                        ;
EOr.       Exp1  ::= Exp1 "||" 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 and underscores, with the following limitations:

- Underscores cannot follow immediately after another.
- An identifier cannot be just an underscore.

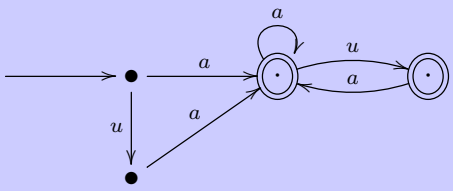
Letters be subsumed under the non-terminal a; besides letters, we only consider underscores u; thus, the alphabet is just $\{a, u\}$.

1. Give a regular expression for identifiers.
2. Give a deterministic finite automaton for identifiers with no more than 7 states.

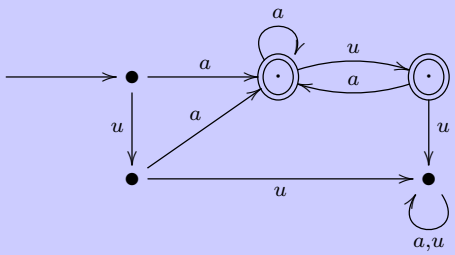
Remember to mark initial and final states appropriately. (4p)

SOLUTION:

1. RE: $u^?(a^+u^?)^+$ which is $(\varepsilon + u)aa^*(\varepsilon + u)(aa^*(\varepsilon + u))^*$
   or: $(a + ua)^+u^?$ which is $(a + ua)(a + ua)^*(\varepsilon + u)$
2. DFA:



   DFA with error state:



Question 3 (LR Parsing): Consider the following labeled BNF-Grammar.

```
Cons.  L ::= L "," L ;
Foo.   L ::= "foo"   ;
Bar.   L ::= "bar"   ;
Doh.   L ::= "doh"   ;
```

We work with the following example string:

```
foo , bar , doh
```

1. The grammar is ambiguous. Show this by giving two different parse trees for the example string.

2. The LR parser generated for this grammar has a shift-reduce conflict.

   Step by step, trace the parsing of the example string showing how the stack and the input evolve and which actions are performed. Resolve any shift-reduce conflict in favor of *shift*.

   Which of the two parse trees is produced by this run?

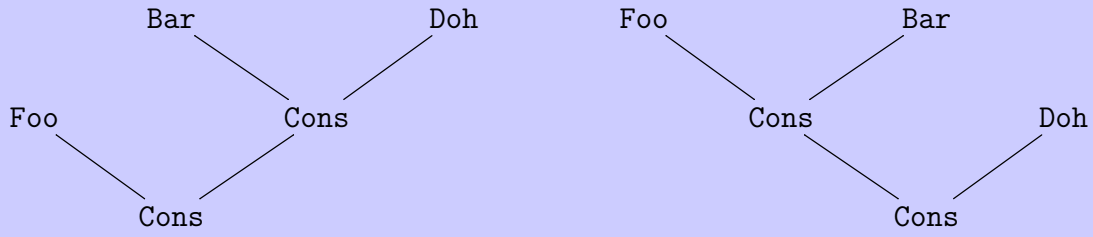
3. Now do the trace again, this time resolving conflicts in favor of *reduce*.

   Which of the two parse trees is produced by that run?

(8p)

SOLUTION:

1. Parsetrees for right associative (shift) and left associative (reduce) reading:



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

```
            . foo , bar , doh   -- shift
foo         .     , bar , doh   -- reduce with rule Foo
L           .     , bar , doh   -- shift 2x
L , bar     .           , doh   -- reduce with rule Bar
L , L       .           , doh   -- **shift** 2x
L , L , doh .                   -- reduce with rule Doh
L , L , L   .                   -- reduce with rule Cons
L , L       .                   -- reduce with rule Cons
L           .                   -- accept
```

   This run produces the first parse tree (right associative).

3. This run differs once we have `L , L` on the stack:

```
L , L   .       , doh   -- **reduce** with rule Cons
L       .       , doh   -- shift 2x
L , doh .               -- reduce with rule Doh
L , L   .               -- reduce with rule Cons
L       .               -- accept
```

   It produces the second parse tree (left associative).

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 context 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".

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

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

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.

   Alternatively, you can write the interpreter in pseudo code or Haskell. Functions `lookupVar` and `updateVar` can be assumed if their behavior is described. In any case, the environment must be made explicit. (6p)

SOLUTION: The evaluation judgement $\gamma \vdash e \Downarrow v$ for expressions is the least

relation closed under the following rules.

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

$$\frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash \texttt{EAss}\ x\ e \Downarrow \langle v; \gamma'[x = v] \rangle} \qquad \frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle i_2; \gamma'' \rangle}{\gamma \vdash \texttt{ELEq}\ e_1\ e_2 \Downarrow \langle i_1 \leq i_2; \gamma'' \rangle}$$

$$\frac{\gamma \vdash e_1 \Downarrow \langle 1; \gamma' \rangle}{\gamma \vdash \texttt{EOr}\ e_1\ e_2 \Downarrow \langle 1; \gamma' \rangle} \qquad \frac{\gamma \vdash e_1 \Downarrow \langle 0; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle b; \gamma'' \rangle}{\gamma \vdash \texttt{EOr}\ e_1\ e_2 \Downarrow \langle b; \gamma'' \rangle}$$

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

Question 5 (Compilation):

1. *Statement by statement*, translate the example program of Question 1 to Jasmin. (Do not optimize the program before translation!)

   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. (9p)

SOLUTION:

```
.method public static f(IZ)Z
.limit locals 4
.limit stack 2

 ;; int y = 127;

 bipush 127
 istore_2

 ;; if (b || (y = 42) <= x)

 iload_1
 ifne L2
 bipush 42
 istore_2
 iload_2
 iload_0
 if_icmple L2

 ;; return x <= 55;
```

```
 iload_0
 bipush 55
 if_icmple L0
 iconst_0
 goto L1
L0:
 iconst_1
L1:
 ireturn
L2:

 ;; bool x = y <= 100;

 iload_2
 bipush 100
 if_icmple L3
 iconst_0
 goto L4
L3:
 iconst_1
L4:
 istore_3

 ;; return x || b;

 iload_3
 ifne L5
 iload_1
 ifne L5
 iconst_0
 goto L6
L5:
 iconst_1
L6:
 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. (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	
`iload` a	(P, V, S)	$\rightarrow$	$(P+1, V, S.V(a))$	
`istore` a	$(P, V, S.v)$	$\rightarrow$	$(P+1, V[a := v], S)$	
`iconst` i	(P, V, S)	$\rightarrow$	$(P+1, V, S.i)$	
`bipush` i	(P, V, S)	$\rightarrow$	$(P+1, V, S.i)$	
`goto` L	(P, V, S)	$\rightarrow$	(L, V, S)	
`ifne` L	$(P, V, S.0)$	$\rightarrow$	$(P+1, V, S)$	
`ifne` L	$(P, V, S.v)$	$\rightarrow$	(L, V, S)	if $v \neq 0$
`if_icmple` L	$(P, V, S.v.w)$	$\rightarrow$	(L, V, S)	if $v \leq w$
`if_icmple` L	$(P, V, S.v.w)$	$\rightarrow$	$(P+1, V, S)$	unless $v \leq w$

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	$::=$	$i \mid e + e \mid x \mid \lambda x \rightarrow e \mid e\, e$	expression
t	$::=$	$\mathsf{Int} \mid t \rightarrow t$	type

Application $e_1\ e_2$ is left-associative, the arrow $t_1 \rightarrow t_2$ 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) $f : (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int}) \vdash (\lambda x \rightarrow f\, x)\, (\lambda f \rightarrow f) : \mathsf{Int} \rightarrow \mathsf{Int}$

(b) $h : (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int} \vdash \lambda y \rightarrow y\, (h\, y) : (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}$

(c) $h : (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int} \vdash \lambda x \rightarrow h\, (h\, x) : (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}$

(d) $z : \mathsf{Int},\ y : \mathsf{Int} \rightarrow \mathsf{Int} \vdash (\lambda f \rightarrow y + 1)\, z : \mathsf{Int}$

(e) $f : \mathsf{Int} \rightarrow \mathsf{Int} \vdash \lambda g \rightarrow f\, (f\, 1 + f\, g) : \mathsf{Int} \rightarrow \mathsf{Int}$

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) valid
(c) not valid
(d) not valid (cannot add 0 to function)
(e) valid

2. For each of the following terms, decide whether it evaluates more efficiently (in the sense of fewer reductions) in call-by-name or call-by-value. Your answer can be just "call-by-name" or "call-by-value", but you can also add a justification why you think so. *Same rules for multiple choice as in part 1.* (5p)

(a) $(\lambda x \to \lambda y \to x + x)\ (1 + 2 + 3 + 4)\ (5 + 6 + 7)$
(b) $(\lambda x \to \lambda y \to y + y)\ (\lambda u \to (\lambda z \to z\, z)(\lambda z \to z\, z))\ (1 + 2)$
(c) $(\lambda x \to x + x)\ ((\lambda y \to \lambda z \to z + z)\ (1 + 2 + 3)\ (4 + 5))$
(d) $(\lambda x \to \lambda y \to y + y)\ ((\lambda z \to z\, z)(\lambda z \to z\, z))\ (1 + 2 + 3)$
(e) $(\lambda x \to \lambda y \to x + x)\ (1 + 2)\ (3 + 4 + 5)$

SOLUTION:

(a) call-by-value (6 additions vs. 7)
(b) call-by-value (2 additions vs. 3)
(c) call-by-value (5 additions vs. 7)
(d) call-by-name (diverges in call-by-value)
(e) call-by-name (3 additions vs. 4)