Programming Language Technology

Exam, 15 January 2026 at 8.30 – 12.30 in HB 1-4

Course codes: Chalmers DAT151, GU DIT231.
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: 28 January 2026 14.30-15.30 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++/Java (sublanguage of lab 2):

- Program: a sequence of function definitions.
- Function definition: type followed by identifier, comma-separated parameter list in parentheses, and 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 = 42;`
  - expression followed by semicolon
  - while statement
- Expressions, from highest to lowest precedence:
  - atoms: identifier, integer literal, function call
  - addition (`+`), left associative
  - less-or-equal-than integer comparison (`<=`), non-associative
  - assignment, right associative

  Putting an expression into parentheses makes it an atom.
- Type: `void` or `int` or `bool`

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

```
#include <stdio.h>
#define printInt(x) printf("%d\n",x)
void fib(int n) {
  int i = 0; int cur = 0; int next = 1;
  while ((i = i + 1) <= n) { int tmp = next; next = tmp + cur; cur = tmp; }
  printInt(cur);
}
```

(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 ";"                   ;
SExp.      Stm   ::= Exp ";"                                  ;
SWhile.    Stm   ::= "while" "(" Exp ")" Stm                  ;
terminator Stm   ""                                           ;

EId.       Exp2  ::= Ident                                    ;
EInt.      Exp2  ::= Integer                                  ;
ECall.     Exp2  ::= Ident "(" [Exp] ")"                      ;
EPlus.     Exp1  ::= Exp1 "+" Exp2                            ;
ELtEq.     Exp   ::= Exp1 "<=" Exp1                           ;
EAss.      Exp   ::= Ident "=" Exp                            ;
coercions  Exp   2                                            ;
separator  Exp   ","                                          ;

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

comment    "#"                                                ;
```

Question 2 (Compilation Phases):

1. List the names of the phases of a compiler in the order they execute in the lab 3 compiler. (1p)
2. Each of the following snippets (from the C– language of lab 2) contains one error. For each of the snippets, write the name of the compilation phase in which this error ought to be detected and reported. (4p)

   (a) `while ((i == i + 1) <= n) { int t = x; x = t + c; c = t; }`

   (b) `while ((i = i + 1) <= n) { int t = x; x = t + ; c = t; }`

   (c) `while ((i = i + 1) <= n) { int t = x; x = t + c; int t = 0; }`

   (d) `while ((i = i "+ 1) <= n) { int t = x; x = t + c; c = t; }`

CLARIFICATION: The variables `i`, `n`, `x` and `c` are assumed to be in scope and of type `int`.

SOLUTION:

1. lexing, parsing, type checking, code generation
2. (a) type checking

   (b) parsing

   (c) type checking

   (d) lexing

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

```
Sub.    E ::= E "-" E ;
One.    E ::= "1"     ;
Two.    E ::= "2"     ;
Three.  E ::= "3"     ;
```

We work with the following example string:

```
3 - 2 - 1
```

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?

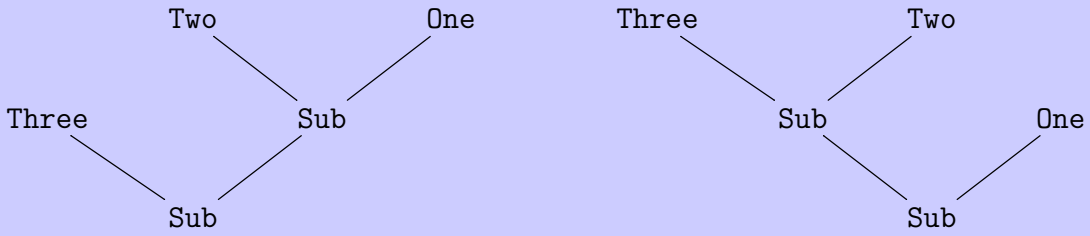
3. Now do the trace again, this time resolving conflicts in favor of *reduce*.

   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.

```
                . '3' '-' '2' '-' '1'  -- shift
'3'             . '-' '2' '-' '1'      -- reduce with rule Three
E               . '-' '2' '-' '1'      -- shift 2x
E '-' '2'       . '-' '1'              -- reduce with rule Two
E '-' E         . '-' '1'              -- *shift* 2x
E '-' E '-' '1' .                      -- reduce with rule One
E '-' E '-' E   .                      -- reduce with rule Sub
E '-' E         .                      -- reduce with rule Sub
E               .                      -- halt
```

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

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

```
E '-' E   .                    -- *reduce* with rule Sub
E         . '-' '1'            -- shift 2x
E '-' '1' .                    -- reduce with rule One
E '-' E   .                    -- reduce with rule Sub
E         .                    -- halt
```

   It produces the second parse tree (left associative).

Question 4 (Type checking and evaluation):

1. Write syntax-directed typing rules for the *expressions* of Question 1. Alternatively, you can write the type-checker in pseudo code or Haskell. Functions `lookupVar` and `lookupFun` can be assumed. In any case, the typing environment must be made explicit. (6p)

SOLUTION: The type checking judgement $\Gamma \vdash_\Sigma e : t$ for expressions is the least relation closed under the following rules.

$$\frac{}{\Gamma \vdash_\Sigma x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash_\Sigma i : \texttt{int}} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \texttt{int} \qquad \Gamma \vdash_\Sigma e_2 : \texttt{int}}{\Gamma \vdash_\Sigma e_1 + e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{int} \qquad \Gamma \vdash_\Sigma e_2 : \texttt{int}}{\Gamma \vdash_\Sigma e_1 \leq e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash_\Sigma e : t}{\Gamma \vdash_\Sigma x = e : t}\ \Gamma(x) = t$$

$$\frac{\Gamma \vdash_\Sigma e_1 : t_1 \quad \ldots \quad \Gamma \vdash_\Sigma e_n : t_n}{\Gamma \vdash_\Sigma f(e_1, \ldots, e_n) : t}\ \Sigma(f) = (t_1, \ldots, t_n) \to t$$

Herein, Γ is a finite map from identifiers x to types t, and Σ a finite map from identifiers f to function types $(t_1, \ldots, t_n) \to t$.

2. Write syntax-directed interpretation rules for the *statements*, *blocks* and *statement sequences* of Question 1, assuming an interpreter for expressions $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$. Alternatively, you can write the interpreter in pseudo code or Haskell. Functions `evalExp` and `lookupVar` can be assumed. In any case, the environment must be made explicit. (7p)

SOLUTION: The evaluation judgement $\gamma \vdash ss \Downarrow \gamma'$ for statement sequences is the least relation closed under the following rules.

$$\frac{}{\gamma \vdash \varepsilon \Downarrow \gamma} \qquad \frac{\gamma \vdash s \Downarrow \gamma' \qquad \gamma' \vdash ss \Downarrow \gamma''}{\gamma \vdash s\ ss \Downarrow \gamma''}$$

The evaluation judgement $\gamma \vdash s \Downarrow r$ for statements with result $r ::= v \mid \gamma'$ is the least relation closed under the following rules.

$$\frac{\gamma. \vdash ss \Downarrow \gamma'.\delta}{\gamma \vdash \{ss\} \Downarrow \gamma'} \qquad \frac{(\gamma, x{=}\mathsf{void}) \vdash e \Downarrow \gamma'}{\gamma \vdash t\ x = e; \Downarrow (\gamma', x{=}v)} \qquad \frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash e; \Downarrow \gamma'}$$

$$\frac{\gamma \vdash e \Downarrow \langle \texttt{false}; \gamma' \rangle}{\gamma \vdash \texttt{while}\ (e)\ s \Downarrow \gamma'}$$

$$\frac{\gamma \vdash e \Downarrow \langle \texttt{true}; \gamma' \rangle \qquad \gamma' \vdash s \Downarrow \gamma'' \qquad \gamma'' \vdash \texttt{while}\ (e)\ s \Downarrow \gamma'''}{\gamma \vdash \texttt{while}\ (e)\ s \Downarrow \gamma'''}$$

Herein, environment γ is a dot-separated list of blocks δ, each of which is a finite map from identifiers x to values v. We write ε for the empty sequence or empty map and $\gamma, x = v$ for extending the top block of γ by the binding $x = v$.

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. (8p)

SOLUTION:

```
.method public static fib(I)V
.limit locals 5
.limit stack 2

 ;; int i = 0;

 iconst_0
 istore_1

 ;; int cur = 0;

 iconst_0
 istore_2

 ;; int next = 1;

 iconst_1
 istore_3

 ;; while ((i = i + 1) <= n)

 goto L1
L0:

 ;; int tmp = next;

 iload_3
 istore 4

 ;; next = tmp + cur;

 iload 4
 iload_2
 iadd
 istore_3
```

```
 ;; cur = tmp;

 iload 4
 istore_2
L1:
 iload_1
 iconst_1
 iadd
 istore_1
 iload_1
 iload_0
 if_icmple L0

 ;; printInt (cur);

 iload_2
 invokestatic Runtime/printInt(I)V
 return

.end method
```

2. Give the small-step semantics of the JVM instructions you used in the Jasmin code in part 1 (except for function call and `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	
`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)	$\rightarrow$	$(P+1, V, S.i)$	
`iadd`	$(P, V, S.v.w)$	$\rightarrow$	$(P+1, V, S.(v+w))$	
`goto` L	(P, V, S)	$\rightarrow$	(L, V, S)	
`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	$::=$	$i \mid e + e \mid x \mid \lambda x \to e \mid e\, e$	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)	$a : \mathsf{Int},\ b : \mathsf{Int} \to \mathsf{Int}$	$\vdash (\lambda c \to 0 + b)\, a$	$: \mathsf{Int}$
(b)	$a : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$	$\vdash \lambda b \to b\, (a\, b)$	$: (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
(c)	$a : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$	$\vdash \lambda b \to a\, (a\, b)$	$: (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
(d)	$a : \mathsf{Int} \to \mathsf{Int}$	$\vdash \lambda b \to a\, (a\, b + a\, 0)$	$: \mathsf{Int} \to \mathsf{Int}$
(e)	$a : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$	$\vdash (\lambda b \to a\, b)\, (\lambda a \to 1)$	$: \mathsf{Int} \to \mathsf{Int}$

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 (cannot add 0 to function)
(b) valid
(c) not valid
(d) valid
(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) $(\lambda x \to \lambda y \to x + x + x)\ (1 + 2 + 3)\ (4 + 5 + 6 + 7)$
(b) $(\lambda x \to \lambda y \to x + x + x)\ (1 + 2)\ (3 + 4 + 5 + 6)$
(c) $(\lambda x \to x)\ ((\lambda y \to \lambda z \to z + z + z)\ (1 + 2 + 3 + 4)\ (5 + 6))$
(d) $(\lambda x \to \lambda y \to y + y)\ ((\lambda z \to z\, z)(\lambda z \to z\, z))\ (1 + 2 + 3)$
(e) $(\lambda x \to \lambda y \to y + y + y)\ (\lambda u \to (\lambda z \to z\, z)(\lambda z \to z\, z))\ (1 + 2)$

SOLUTION:

(a) call-by-value (7 additions vs. 8)
(b) call-by-name (5 additions vs. 6)
(c) call-by-name (5 additions vs. 6)
(d) call-by-name (diverges in call-by-value)
(e) call-by-value (3 additions vs. 5)