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)

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)

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)

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)

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)

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)

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)

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) $f : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int}) \vdash (\lambda x \to f\, x)\, (\lambda f \to f) : \mathsf{Int} \to \mathsf{Int}$

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

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

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

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

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)

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) $(\lambda x \to \lambda y \to x + x)\ (1 + 2)\ (3 + 4 + 5)$

Good luck!