Programming Language Technology

Exam, 12 April 2022 at 08.30 – 12.30 in M

Course codes: Chalmers DAT151, GU DIT231.
Exam supervision: Andreas Abel (+46 31 772 1731), visits at 09:30 and 11:30.

Grading scale: Max = 60p, VG = 5 = 48p, 4 = 36p, G = 3 = 24p.
Allowed aid: an English dictionary.
Exam review: 28 April 2022 13.30-14.30 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++ (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, identifier, integer literal
  - addition (`+`), left associative
  - less-than comparison (`<`), non-associative
  - short-circuiting conjunction (`&&`), left associative
- Type: `int` or `bool`

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 x, bool b) {
  int z = x + x;
  if (b && z < 10) {
    int x = z + z;
    return x;
  } else return 0;
}
```

(10p)

Question 2 (Lexing): An *non-nested Haskell 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 distinguishing 4 tokens: `{`, `}`, `-`, and c where c stands for *any other character*. (6p)

Question 3 (LR Parsing): Use your grammar from Question 1. Step by step, trace the shift-reduce parsing of the expression `b && z < 10` showing how the stack and the input evolve and which actions are performed. (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 environment 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 $\gamma \vdash e \Downarrow v$ where e denotes the expression to be evaluated in environment γ and v the resulting value.

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

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)

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)

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)	$x : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$	$\vdash \lambda y \to y\,(x\,y)$	$: (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
(b)	$f : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int})$	$\vdash (\lambda x \to f\,x)\,(\lambda f \to f)$	$: \mathsf{Int} \to \mathsf{Int}$
(c)	$f : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$	$\vdash \lambda x \to f\,(f\,x)$	$: (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
(d)	$g : \mathsf{Int} \to \mathsf{Int}$	$\vdash \lambda x \to g\,(g\,x + g\,1)$	$: \mathsf{Int} \to \mathsf{Int}$
(e)	$x : \mathsf{Int},\ g : \mathsf{Int} \to \mathsf{Int}$	$\vdash (\lambda y \to g + 0)\,x$	$: \mathsf{Int}$

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. Write a **call-by-name** interpreter for the functional language above, either with inference rules or in pseudo code or Haskell. (5p)