Programming Language Technology

Exam, 13 January 2022 at 08.30 – 12.30 in HA1-4

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: 24 January 2022 13.30-15.00 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++:

- Program: `int main()` followed by a block
- Block: a sequence of statements enclosed between `{` and `}`
- Statements:
  - block
  - variable declaration, e.g., `int x;`
  - statement formed from an expression by adding a semicolon `;`
  - `while` statement
- Expressions, from highest to lowest precedence:
  - parenthesized expression, identifier, integer literal
  - addition (`+`), left associative
  - less-than comparison (`<`), non-associative
  - assignment (`x = e`), right associative
- Type: `int` or `bool`

You can use the standard BNFC categories `Integer` and `Ident` but *none* of the BNFC pragmas (`coercions`, `terminator`, `separator` ...). An example program is:

```
int main () {
  int x;  x = 0;
  while ((x = x + 1) < 10) int x;
}
```

(10p)

Question 2 (Lexing): An *identifier* be a non-empty sequence of letters, digits and underscores, with the following limitation: *There must be at least one letter between any two of these positions: beginning, end, and any underscore position.* In other terms, if you cut the identifier into words at each underscore, each of the words needs to contain at least one letter.

Letters be subsumed under the non-terminal l and *digits* under d, thus, the alphabet is just $\{l, d, \_\}$.

1. Give a regular expression for identifiers.

2. Give a deterministic finite automaton for identifiers with no more than 9 states.

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

Question 3 (LR Parsing): Consider the following labeled BNF-Grammar. The starting non-terminal is `C`.

```
Cond.  C ::= S "<" S ;
Sum.   S ::= S "+" X ;
Atom.  S ::= X       ;

A.     X ::= "a"     ;
B.     X ::= "b"     ;
C.     X ::= "c"     ;
D.     X ::= "d"     ;
```

Step by step, trace the shift-reduce parsing of the expression

```
a + b < c + d
```

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 $\Gamma \vdash s \Rightarrow \Gamma'$ where s is a statement or list of statements, 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 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. 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. Write compilation schemes in pseudo code or Haskell for the statement, block, and expressions constructions of Question 1. The compiler should output symbolic JVM instructions (i.e. Jasmin assembler). It is not necessary to remember exactly the names of the instructions—only what arguments they take and how they work.

   Service functions like `addVar`, `lookupVar`, `lookupFun`, `newLabel`, `newBlock`, `popBlock`, and `emit` can be assumed if their behavior is described. (9p)

2. Give the small-step semantics of the JVM instructions you used in the compilation schemes in part 1. 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.

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

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)

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