

Programming Language Technology

Exam, 13 January 2020 at 08.30 – 12.30 in M

Course codes: Chalmers DAT151, GU DIT231. As re-exam, also DAT150 and DIT230.
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 2019 13.30-15.00 in EDIT meeting room *Analysen* (3rd floor).

Please answer the questions in English.

Question 1 (Grammars): Write a labelled BNF grammar that covers the following kinds of constructs of C:

- Program: `int main()` followed by a block
- Block: a sequence of statements enclosed between `{` and `}`
- Statement:
 - block
 - initializing variable declaration, e.g., `int x = e;`
 - statement formed from an expression by adding a semicolon `;`
 - `if` statement with `else`
- Expression:
 - boolean literal `true` or `false`
 - integer literal
 - function call with a list of comma-separated arguments
 - post-increment of an identifier, e.g., `x++`
 - addition (+), left associative
 - parenthesized expression
- Type: `int` or `bool`

Lines starting with `#` are comments. An example program is:

```
#include <stdio.h>
#define printInt3(e1,e2,e3) printf("%d %d %d\n",e1,e2,e3)
int main () {
    int x = 8;
    if (true) {
        printInt3 (x++, 10 + x++, x++ + 19);
    } else bool b = false;
}
```

You can use the standard BNFC categories `Integer` and `Ident`, the `coercions` pragma, and list categories via the `terminator` and `separator` pragmas.

(10p)

Question 2 (Lexing): An *acceptable password* be a sequence of characters that contains at least one digit and one special character. Our alphabet be $\Sigma = \{a, b, c\}$ where a stands for digits, b for special characters, and c for other characters (like letters).

1. Give a regular expression for acceptable passwords.
2. Give a deterministic finite automaton for acceptable passwords with no more than 8 states.

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

Question 3 (LR Parsing): Consider the following labeled BNF-Grammar (written in `bnfc` syntax). The starting non-terminal is `S`.

```
Seq.   S ::= P M      ;
Plus.  P ::= P A "+"  ;
None.  P ::=           ;

Minus. M ::= A "-" M  ;
Done.  M ::= A        ;

X.     A ::= "x"      ;
Y.     A ::= "y"      ;
```

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

`x + y - x`

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. You can assume a judgement $\gamma \vdash b \Downarrow v$ stating that block b evaluates to value v in environment γ .

Alternatively, you can write the interpreter in pseudo code or Haskell (then assume a function `evalBlock` to be defined). 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
$n ::= 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \dots$		numeral
$e ::= n \mid e + e \mid x \mid \lambda x \rightarrow e \mid e e$		expression
$t ::= \text{Int} \mid t \rightarrow t$		type

Application $e_1 e_2$ is left-associative, the arrow $t_1 \rightarrow t_2$ 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) $\vdash \lambda x \rightarrow \lambda y \rightarrow (y x) 0$: $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
- (b) $g : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \vdash (g + 1) (\lambda x \rightarrow x)$: Int
- (c) $f : \text{Int} \rightarrow \text{Int} \vdash \lambda x \rightarrow f (f (1 + (f x)))$: $\text{Int} \rightarrow \text{Int}$
- (d) $x : \text{Int} \rightarrow \text{Int}, g : \text{Int} \vdash x (g + 1)$: Int
- (e) $f : (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) \vdash (\lambda x \rightarrow f x) (\lambda x \rightarrow f (\lambda x \rightarrow x) x)$: $\text{Int} \rightarrow \text{Int}$

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)