Programming Language Technology

Exam, 24 August 2017 at 14.00–18.00 in M

Course codes: Chalmers DAT151, GU DIT231. As re-exam, also DAT150, DIT229/230, and TIN321.
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: Tuesday 12 September 2017 at 13.30 in room EDIT 8103 (past the CSE lunchroom).

Please answer the questions in English.

Question 1 (Grammars): Write a labelled BNF grammar that covers the following constructs of a C-like imperative language: A program is a list of statements. Types are `int` and `bool`. Statement constructs are:

- variable declarations (e.g. `int` x`;`), not multiple variables, no initial value
- expression statements (E`;`)
- `while` loops
- blocks: (possibly empty) lists of statements enclosed in braces

Expression constructs are:

- identifiers/variables
- integer literals
- post-increments of *identifiers* (x`++`)
- less-or-equal-than comparisons (E `<=` E')
- assignments of identifiers (x `=` E)

Less-or-equal is non-associative and binds stronger than assignment. Parentheses around and expression are allowed and have the usual meaning. An example program would be:

```
int x; x = 0; while (x++ <= 9) {}
```

You can use the standard BNFC categories `Integer` and `Ident` as well as list short-hands, and `terminator`, `separator`, and `coercions` rules. (10p)

Question 2 (Type checking and evaluation):

1. Write syntax-directed *type checking* rules for the *statement* forms and lists of Question 1. The typing environment must be made explicit. You can assume a type-checking judgement for expressions.

   Alternatively, you can write the type-checker in pseudo code or Haskell.

   Please pay attention to scoping details; in particular, the program

   ```
   while (0 <= 1) int x; x = 0;
   ```

   should not pass your type checker! (5p)

2. Write syntax-directed *interpretation* rules for the *expression* forms of Question 1. The environment must be made explicit, as well as all possible side effects.

   Alternatively, you maybe write an interpeter in pseudo code or Haskell. (5p)

Question 3 (Compilation):

1. Write compilation schemes in pseudo code for each of the *expression* constructions in Question 1 generating JVM (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. (6p)

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) are 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 4 (Regular Languages): Company *SaniSol* develops showers and has bought a water-proof robot from company *RoboCRP* for testing its newest shower models. The testing environment consists of two adjacent square rooms separated by a swing door. Room 1 is empty, except for the swing door to room 2. Room 2 contains the shower (and of course the swing door back to room 1). *RoboCRP* has programmed the test robot with two actions.

- *a* *Move forward through the swing door and spin by 180°.* This action can be carried out whenever the robot faces a door into another room.
- *b* *Take a shower, spinning by 360°.* This action can be carried out whenever the robot is in a room with a shower.

If the robot is asked to perform an action it cannot carry out, it will explode according to the *RoboCRP SelfDestruct* ® mechanism.

In the beginning, the robot is in room 1 facing the swing door to room 2. A *valid action sequence* is a non-empty sequence of *a* and/or *b* actions that does not make the robot explode and returns it to room 1 in the end. For example, the sequences *abbba* and *aaabbaaba* are valid and *aaa*, *ab*, and *ba* are invalid.

1. Give a regular expression for valid action sequences. Demonstrate that your regular expression accepts the two valid examples and rejects the three invalid ones. (5p)
2. Give a deterministic or non-deterministic automaton for recognizing valid action sequences. Demonstrate that your automaton accepts the two valid examples and rejects the three invalid ones. (5p)

Question 5 (Parsing): Consider the following LBNF-Grammar for arithmetical expressions (written in `bnfc`). The starting non-terminal is `S`.

```
Plus.     S ::= S "+" P   ; -- Sums
Product.  S ::= P         ;

Times.    P ::= P "*" A   ; -- Products
Atom.     P ::= A         ;

X.        A ::= "x"       ; -- Atoms
Y.        A ::= "y"       ;
Z.        A ::= "z"       ;
Parens.   A ::= "(" S ")" ;
```

Step by step, trace the LR-parsing of the expression

```
x + y * z
```

showing how the stack and the input evolves and which actions are performed. For each reduce action, mention the grammar rule used to reduce the stack. (8p)

Question 6 (Functional languages):

1. For lambda-calculus expressions we use the abstract grammar

$$e ::= n \mid x \mid \lambda x \to e \mid e\, e$$

   and for simple types $t ::= \mathbb{N} \mid t \to t$. Non-terminal x ranges over variable names and n over non-negative integer constants 0, 1, etc.

   For the following typing judgements $\Gamma \vdash e : t$, decide whether they are valid or not. Your answer should be just "valid" or "not valid".

   (a) $y : \mathbb{N} \to \mathbb{N}, f : \mathbb{N} \vdash f\, y : \mathbb{N}$.

   (b) $y : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \vdash y\, (\lambda x \to 1) : \mathbb{N}$.

   (c) $f : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \vdash (\lambda x \to f\, (x\, x))\, (\lambda x \to f\, (x\, x)) : \mathbb{N} \to \mathbb{N}$.

   (d) $\vdash \lambda x \to \lambda y \to (f\, x)\, y : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$.

   (e) $f : \mathbb{N} \to \mathbb{N} \vdash \lambda x \to f\, (f\, x) : \mathbb{N} \to \mathbb{N}$.

   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 above lambda-calculus either with inference rules, or in pseudo-code or Haskell. (5p)

Good luck!