Programming Language Technology

Exam, 08 April 2021 at 08.30 – 12.30 on Canvas

Course codes: Chalmers DAT151, GU DIT231. As re-exam, also DAT150 and DIT230.
Exam supervision: Andreas Abel. Questions may be asked in Zoom breakout room, by email (`mailto:andreas.abel@gu.se`, subject: **PLT exam**) or telephone (+46 31 772 1731).

Exam review: Modalities will be announced later.
Allowed aids:

- All exam questions have to be solved *individually*.
- *No communication* of any form is permitted during the exam, including conversation, telephone, email, chat, asking questions in internet fora etc.
- All course materials can be used, including the book, lecture notes, previous exam solutions, own lab solution, etc. Any material copied verbatim should be marked as *quotation with reference* to the source.
- Publicly available *documentation* on the internet may be consulted freely to prepare the solution. *Small* portions of code and text from publicly available resources may be reused in the solution if *clearly marked* as quotation and *properly referencing* the source.

Any violation of the above rules and further common sense rules applicable to an examination, including *plagiarism* or *sharing solutions* with others, will lead to immediate failure of the exam (grade U), and may be subject to further persecution.
Grading scale: VG = 5, G = 4/3, U.

To pass, you need to deliver complete answers to two out of questions 1-3. (Typos, bugs, and minor omissions are not a problem as long as your answer demonstrates good understanding of the subject matter.) For a Chalmers grade 4 you need complete answers to all of the questions 1-3. A VG/5 requires excellent answers on questions 1-3.

Submission instructions:

- Please answer the questions in English.
- The solutions need to be submitted as one `.zip` archive, named according to schema **`FirstName_LastName_Personnummer.zip`**. Checklist:
  - `Lovelace.cf`
  - `Sum.adb`
  - `Question2.{txt|md|pdf|...}`
  - `Question3.{txt|md|pdf|...}`
  - (other relevant files)

In the following, a fragment *Lovelace* of the Ada programming language is described, in its syntax and semantics. Two example programs, `Primes.adb` and `Factorial.adb` are included to clarify the specification. In the exam, you are asked to describe a grammar, a type checker, and a compiler for Lovelace.

1. A *program* consists of:

   (a) imports, in Lovelace fixed to the two lines

   ```
   with Ada.Integer_Text_IO;
   use  Ada.Integer_Text_IO;
   ```

   (b) main header: `procedure` *identifier* `is`

   (c) a list of *function definitions*,

   (d) a list of main *variable declarations*,

   (e) a main *block*.

   Running a program will execute the variable declarations and the statements of the block (from which functions can be called).

2. A *variable declaration* is a non-empty comma-separated list of identifiers followed by a colon, a *type*, colon-equals, an initializing expression, and a semicolon. The scope of the initializing expression are the functions and variables declared before, *not* including the variable(s) we are just initializing. Note: If we declare several variables, the initializing expression is evaluated again *for each* variable.

3. A *type* is `Integer` or `Boolean`.

4. A *function definition* consists of:

   (a) header: `function` *identifier* parenthesized-*parameters* `return` *type* `is`,

   (b) a list of local *variable declarations*,

   (c) body: a *block* for the function, terminated by a semicolon.

   The *parameters* are a non-empty semicolon-separated list of *parameter declarations* each of which consists of: *identifier* colon *type*.

   A function needs to be called (see *function call* expression) with the correct number of arguments of the correct type. The call will execute the block with parameters initialized to their respective argument value and local variables initialized to their value (see 2.). The execution of the function ends when a `return` statement is encountered.

   The joint list of parameters and local variables may not have any duplicates.

5. A *block* for a function or procedure with name *identifier* is started by keyword `begin` and ended by `end` *identifier* semicolon. In between is a non-empty list of *statements*, each terminated by a semicolon.

6. A statement can be one of the following. The typing and execution of the statements is like in C/C++/Java unless noted otherwise.

(a) A return statement: `return` *expression*. Returns from the current function with the value of the *expression*.

(b) An assignment: *identifier* colon-equals *expression*.

(c) A conditional: `if` *expression* `then` *statements*, optionally followed by `else` *statements*, terminated by `end if`.

(d) A while-loop: `while` *expression* `loop` *statements* `end loop`.

(e) A for-loop: `for` *identifier* `in` *expression* dot-dot *expression* `loop` *statements* `end loop`. The for-loop declares a new variable *identifier* of type `Integer`, the so-called loop variable. This variable is only in scope in the *statements* and it may shadow other variables. The first expression denotes the initial value of the loop variable and the second expression the final value. Both values are integers and computed before the loop starts. If the final value is below the initial value, the loop is not executed. Otherwise, the loop variable is set to the initial value. The statement is executed, and the loop variable is incremented by one. The actions of the previous sentence are repeated as long as the loop variable is not larger than the final value.

(f) A print statement: `put` followed by a parenthesized expression of type `Integer`. Prints the value and a newline character to the standard output.

7. An expression can be one of the following. Typing and interpretation of expressions is like in C/C++/Java unless noted otherwise.

(a) A variable: *identifier*.

(b) A boolean constant `true` or `false` ~~or an *integer literal*~~ .

(c) An *integer literal*.

(d) A *function call*: *identifier* followed by a parenthesized non-empty comma-separated list of *expressions*.

(e) A parenthesized expression.

(f) A infix binary operation: *expression operator expression*. All operators are left associative. Operators come in four binding strengths:

i. Multiplicative operators, bind strongest:
   - integer multiplication `*`,
   - integer division `div`,
   - integer remainder `mod`.

ii. Additive operators, next in binding strength:
   - integer addition `+`,
   - integer subtraction `-`.

iii. Relational operators, but-last in strength: Equality operators `=` (equal) and `/=` (not equal) and integer comparison operators `<`, `<=`, `>`, and `>=` with the usual meaning.

iv. Short-circuiting logical operators, least in binding strength:

v. boolean conjunction `and then`,

vi. boolean disjunction `or else`.

Operators are always applied to two expression of the same type. Equality operators apply to booleans and to integers. Like in C/C++/Java, boolean conjunction and disjunction are short-circuiting, i.e., if the left operand determines the value of the operation, the right operand is not evaluated.

8. An *identifier* starts with a letter, followed by a possibly empty sequence of letters, digits, and underscores. (Note: this is different from BNFC's `Ident` token type.)

9. An *integer literal* is a non-empty sequence of digits.

Comments start with double-dash (`--`) and last until the end of the line.

An identifier is *never* in scope before its declaration. The detailed scoping rules are:

1. Functions are in scope *after* their declaration: in their own body, in functions defined later, and in the main block. There is no mutual recursion. All functions must have distinct names.

2. The parameters and local variables of a function must be distinct. They are only in scope in the corresponding initializing expressions (see above) and the function body. They may shadow function identifiers.

3. The main variables (as well as all functions) are in scope in the main block. The names of the main variables must be distinct from each other and from the functions.

```
-- Factorial.adb

with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;

procedure Factorial is

   function factorial (n : Integer) return Integer is
   begin
      if n < 2 then
         return 1;
      else
         return n * factorial(n - 1);
      end if;
   end factorial;

   n : Integer := 7;
begin
   put(factorial(n));
end Factorial;
```

```
-- Primes.adb

with Ada.Integer_Text_IO;
use  Ada.Integer_Text_IO;

procedure Primes is

   function prime (n : Integer) return Boolean is
      i : Integer := 3;
   begin
      if n <= 2         then return (n = 2); end if;
      if n mod 2 = 0    then return false;   end if;
      while i * i <= n  loop
         if n mod i = 0 then return false;   end if;
         i := i + 2;
      end loop;
      return true;
   end prime;

   -- Test 100 numbers for primality, starting with 1.
   lower : Integer := 1;
   upper : Integer := 100 + lower - 1;

begin
   for n in lower .. upper loop
      if prime(n) then
         put(n);
      end if;
   end loop;
end Primes;
```

Question 1 (Grammar)

1. Write an Lovelace program `Sum.adb` that computes and prints the sum of the integers from 1 to 100. This program should contain a function `sum` with two integer parameters determining the range (e.g. "from 1 to 100"), and the main block should call this function with arguments 1 and 100.

2. Write a labelled BNF grammar for Lovelace in a file `Lovelace.cf` and create a parser from this grammar using BNFC. For the best evaluation, the parser should be free of conflicts (shift/reduce and reduce/reduce).

3. Recommended: Test your parser on `Primes.adb`, `Factorial.adb` and `Sum.adb`.

Deliverables: files `Lovelace.cf` and `Sum.adb`.

Question 2 (Type checker): Write a specification of a type checker for the Lovelace language of Question 1. The type checker receives an abstract syntax tree of a Lovelace program and shall throw an error if any of the scoping or typing rules are violated.

Deliverable: **submit a text document** with name `Question2` (plus file extension) that contains the specification. The text document can be a plain text file possibly using markup (like markdown) or a PDF.

The specification should have the following structure:

A. State. Describe the components of the *state* of the type checker and how these components are implemented, i.e., which data structure (like list, map, integer...) is used for each component.

B. Initialization and run: Describe how the state is initialized and how the type checker (C) is started (i.e., which arguments are given to the type checker).

C. Syntax-directed traversal: Describe the type checker: Write an explanation how each relevant Lovelace construct (expression, statement, block, declaration, ...) is checked (or its type inferred). You may use judgements and rules or pseudo-code or *precise* language.

D. API (optional): If you used helper functions to manipulate the state in item C, describe them here.

The specification can use the names from your BNFC grammar.

The specification should be written in a high-level but self-contained way so that an *informed outsider* can implement the type checker easily following your specification. An informed outsider shall be a person who has very good programming skills and good familiarity with programming language technology in general, but no specific knowledge about the Lovelace language nor access to the course material.

The specification will be judged on clarity and correctness.

Question 3 (Compilation): Specify a compiler from Lovelace to JVM. The compiler takes a type-correct abstract syntax tree of a Lovelace program as input and translates this into Jasmin method definitions which are printed to the standard output.

Deliverable: **submit a text document** with name `Question3` (plus file extension) that contains the specification. Instructions analogous to Question 2 apply. In particular, follow the same structure: A. State, B. Initialization and run, C. Compilation schemes, D. API.

Restrictions of the task:

1. The compiler does not have to output a full Jasmin class file, only the methods corresponding to the defined Lovelace functions and a `main` method for the main block. (You may assume that no Lovelace function is called `main`.)
2. You need not output `.limit` pragmas (stack/locals).
3. You may simply use the Lovelace function identifiers for the corresponding Jasmin method names.
4. You can assume a Java method that can be called to output an integer.
5. You need not care about Java modifiers like `public` or `static`.
6. It is sufficient to treat one logical, one arithmetical, and one comparison operator.
7. Choose *one* of `if-then` or `if-then-else` or `while`.

However, the compiler needs to output proper JVM instructions (not pseudo machine code).
Good luck!