Programming Language Technology

Exam, 11 January 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:
  - `Blaise.cf`
  - `Sum.pas`
  - `Question2.{txt|md|pdf|...}`
  - `Question3.{txt|md|pdf|...}`
  - (other relevant files)

In the following, a fragment *Blaise* of the Pascal programming language is described, in its syntax and semantics. Two example Blaise programs, `Primes.pas` and `Factorial.pas` are included to clarify the specification. In the exam, you are asked to describe a grammar, an interpreter, and a compiler for Blaise.

1. A *program* consists of:

   (a) header: **`program`** *identifier* semicolon,

   (b) a list of *function definitions*,

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

   (d) a main *block*, terminated by a dot.

   Running a program will execute the statements of the block (from which functions can be called). The name of the program (given by the *identifier*) is ignored.

2. A *variable declaration* starts with **`var`** followed by a non-empty comma-separated list of identifiers, a colon, a *type*, and a semicolon. Declared variables are implicitly *initialized* to the default value of their type: integer variables are initialized to `0` and boolean variables to `false`.

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

4. A *function definition* consists of:

   (a) header: **`function`** *identifier* parenthesized-*parameters* colon *type* semicolon,

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

   (c) body: a *block*, 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 default value (see above). The local variables contain an additional *result variable* that bears the name of the function. This result variable can be read and assigned like an ordinary variable or parameter. When the execution of the block ends, the function will return the content of this result variable.

   The joint list of parameters and local variables (including the result variable) may not have any duplicates.

5. A *block* is delimited by keywords **`begin`** and **`end`** and contains a possibly empty semicolon-separated list of *statements*.

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) The empty statement, does nothing.

   (b) A *block*.

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

(d) A conditional: `if` *expression* `then` *statement* `else` *statement*.

(e) A while-loop: `while` *expression* `do` *statement*.

(f) A for-loop: `for` *identifier* colon-equals *expression* `to` *expression* `do` *statement*. The identifier is the loop variable which must be in scope and be of type `Integer`. 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 and the loop variable not set. 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.

(g) A print statement: `writeln` followed by a parenthesized integer expression. 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) A *function call*: *identifier* followed by a parenthesized non-empty comma-separated list of *expressions*.

   (d) A parenthesized expression.

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

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

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

      iii. Relational operators, least in binding strength: Equality operators = (equal) and <> (not equal) and integer comparison operators <, <=, >, and >= with the usual meaning.

   Operators are always applied to two expression of the same type, there is no coercion. Equality operators apply to booleans and to integers. Like in C/C++/Java, boolean conjunction and disjunction are short-circuting, 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 or underscore, 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.

Block comments start with `(*` and end with `*)`.

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.

2. The parameters, local variables, and result variable of a function are only in scope in the function body.

3. The main variables (as well as all functions) are in scope in the main block.

CLARIFICATION: Formulation "there is no coercion" applies to type conversion and should not be confused with the BNFC `coercions` pragma.

```
(* Primes.pas *)

program Primes;

  function prime (n : Integer): Boolean;
  var i : Integer;
  begin
     if n <= 2 then
        prime := (n = 2)
     else begin
        prime := (n mod 2 <> 0);
        i := 3;
        while prime and (i * i <= n) do begin
           prime := (n mod i <> 0);
           i := i + 2;
        end;
     end;
  end;

var lower, upper : Integer;
var n : Integer;
begin
   (* Primes from 1 to 100: *)
   lower := 1;
   upper := 100;
   for n := lower to upper do
      if prime(n) then writeln(n) else;
end.
```

```
(* Factorial.pas *)

program _;

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

var n : Integer;
begin
   n := 7;
   writeln (factorial(n));
end.
```

CLARIFICATION: One example should have a proper case for `separator Stm` instead `terminator`. (E.g. omit semicolon before `end`.)

Question 1 (Grammar)

1. Write an Blaise program `Sum.pas` 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 Blaise in a file `Blaise.cf` and create a parser from this grammar using BNFC. The parser should be free of conflicts (shift/reduce and reduce/reduce).
3. Recommended: Test your parser on `Primes.pas`, `Factorial.pas` and `Sum.pas`.

Deliverables: files `Blaise.cf` and `Sum.pas`.

Question 2 (Interpretation): Write a specification of an interpreter for the Blaise language of Question 1. The interpreter receives a type-correct abstract syntax tree of a Blaise program and produces the output of this program that is generated by the `writeln` statements.

CLARIFICATION: type-correct and well-scoped AST

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 interpreter and how these components are implemented, i.e., which data structure (like list, map, integer...) is used for each component. If the parts of the interpreter produce values, describe their form.

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

C. Interpretation: Describe the interpreter: Write an explanation how each relevant Blaise construct (expression, statement, block, declaration, ...) is evaluated or executed. You may use judgements and rules or pseudo-code or precise language.

   Restriction: You need not describe all of the binary operators. It is sufficient to cover:

   (a) one logical operator (`and` or `or`),

   (b) one arithmetical operator (`+`, `-`, `*`, `div`, or `mod`), and

   (c) one comparison operator (=, <>, <, <=, >, or >=).

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

The specification should be written in a high-level but self-contained way so that an *informed outsider* can implement the interpreter 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 Blaise language nor access to the course material.

The specification will be judged on clarity and correctness.

CLARIFICATION: The specification can use the names from your BNFC grammar.

Question 3 (Compilation): Specify a compiler from Blaise to JVM. The compiler takes a type-correct abstract syntax tree of a Blaise 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 Blaise functions and a `main` method for the main block. (You may assume that no Blaise function is called `main`.)
2. You need not output `.limit` pragmas (stack/locals).
3. You may simply use the Blaise function identifiers for the corresponding Jasmin method names.
4. You need not care about Java modifiers like `public` or `static`.
5. As in Question 2, it is sufficient to treat one logical, one arithmetical, and one comparison operator.
6. Choose *one* of `if` or `while`.

CLARIFICATION: You can assume a Java method `writeln` that can be called to output an integer.

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

Good luck!