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 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 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 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 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 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.
```

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`.

SOLUTION: Summation program (file `Sum.pas`):

```
(* Sum.pas *)

program _;

  function sum (lower : Integer; upper : Integer): Integer;
  var i : Integer;
  begin
     sum := 0;
     for i := lower to upper do sum := sum + i;
  end;

begin
   writeln(sum(1,100));
end.
```

Grammar (file `Blaise.cf`):

```
-- BNFC Grammar of Blaise, a fragment of Pascal

Prg.       Program ::= "program" Id ";" [FunDef] Body ".";
terminator FunDef ";";

Bdy.       Body ::= [VarDecl] Block;
terminator VarDecl ";";

-- # Declarations

FunDf.     FunDef ::= "function" Id "(" [ParDecl] ")" ":" Type ";" Body;
separator  nonempty ParDecl ";";

ParDcl.    ParDecl ::= Id ":" Type;

VarDcl.    VarDecl ::= "var" [Id] ":" Type;
separator  nonempty Id ",";

-- # Types

TInt.      Type ::= "Integer";
TBool.     Type ::= "Boolean";

internal
  TVoid.   Type ::= "Void";

-- # Blocks

Blck.      Block ::= "begin" [Stm] "end";
separator  nonempty Stm ";";

-- # Statements

SBlock.    Stm ::= Block;
SAssign.   Stm ::= Id ":=" Exp;
SIfElse.   Stm ::= "if" Exp "then" Stm "else" Stm;
SWhile.    Stm ::= "while" Exp "do" Stm;
SFor.      Stm ::= "for" Id ":=" Exp "to" Exp "do" Stm;
SWriteln.  Stm ::= "writeln" "(" Exp ")";
SEmpty.    Stm ::= "";

-- # Expressions

EVar.      Exp3 ::= Id;
ETrue.     Exp3 ::= "true";
EFalse.    Exp3 ::= "false";
EInt.      Exp3 ::= Integer;
```

```
ECall.      Exp3 ::= Id "(" [Exp] ")";
EMul.       Exp2 ::= Exp2 MulOp Exp3;
EAdd.       Exp1 ::= Exp1 AddOp Exp2;
ECmp.       Exp  ::= Exp  CmpOp Exp1;

coercions  Exp 3;
separator  nonempty Exp ",";

-- # Operators

OTimes.     MulOp ::= "*"  ;
ODiv.       MulOp ::= "div";
OMod.       MulOp ::= "mod";
OAnd.       MulOp ::= "and";

OPlus.      AddOp ::= "+"  ;
OMinus.     AddOp ::= "-"  ;
OOr.        AddOp ::= "or" ;

OLt.        CmpOp ::= "<"  ;
OLtEq.      CmpOp ::= "<=" ;
OGt.        CmpOp ::= ">"  ;
OGtEq.      CmpOp ::= ">=" ;
OEq.        CmpOp ::= "="  ;
ONEq.       CmpOp ::= "<>" ;

-- # Identifiers

token       Id (letter | '_') (letter | digit | '_')*;

-- # Comments

comment     "(*" "*)";
```

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 in your BNFC grammar.

SOLUTION:

A. State

The state of the interpreter has two following components:

- Signature `sig`: a finite map from identifiers to function definitions (`FunDef`).
- Environment `env`: a finite map from identifier to values.

A value is either (tagged union) a boolean or a integer literal.

B. Initialization and run

The interpreter receives a `Program` consisting of a list of function definitions, a list of variable declarations, and a main block. The function definitions are turned into a map indexed by the function identifier and stored in `sig`. The variable declarations are turned into a map sending the variable identifier to the default value of its type: `false` for `Boolean` and `0` for `Integer`. This map is stored in `env`. Then the interpreter for blocks is called on the main block.

C. Interpreter

The interpreter is a collection of mutually recursive procedures and functions that execute blocks and statements and evaluate expressions.

Execution of blocks, lists of statements, and statements changes `env` and may print to standard output:

- Block: execute its statements.
- List of statements: in first-to-last order, execute each statement.
- Empty statement: no change.
- Print statement `writeln`(e): Evaluate expression e to an integer literal i and print this to standard output.
- Assignment $x := e$: evaluate the expression e to a value v and *assign* x to v, i.e., update `env` so that x is mapped to v.
- Conditional `if` e `then` s_1 `else` s_2: Evaluate e to a boolean literal. If it is `true`, execute s_1, otherwise s_2.
- Loop `while` e `do` s: Evaluate e to a boolean literal. If it is `true`, execute s followed by `while` e `do` s. Otherwise, do nothing.
- Loop `for` $x := e_1$ `to` e_2 `do` s: Evaluate e_1 to integer i_1 and e_2 to i_2. If $i_2 < i_1$, do nothing. Otherwise, assign x to i_1 and repeat the following until the value of x is greater than i_2: Execute s and increase the value of x by 1.

Interpretation of an expression produces a value but does not change `env`.

- Literals evaluate to themselves.
- A variable x evaluates to its value in `env`.
- A call to function f with arguments $e_1, \ldots, e_n$ evaluates as follows: First, evaluate the arguments e_i to values v_i and save `env`. Then, look up f in `sig` yielding a parameter list, a return type t, a local variable list and a block. We prepare a new environment `env` mapping each parameter to its value v_i, each local variable to the default value of its type and identifier f to the default value of t. In this new environment, we execute the block. The value of f in the resulting `env` is the value of the function call, which we return after restoring `env`.
- For logical conjunction e_1 `and` e_2, we evaluate e_1 to a boolean literal. If it is `false`, we return `false`, otherwise we return the value of e_2. (Disjunction `or` is evaluated analogously, short-circuiting on `true`.)
- For arithmetic operations e_1 *op* e_2, we evaluate e_1 and e_2 to integer literals i_1 and i_2 and return the result of applying *op* to these literals.
- For integer comparison operations e_1 *op* e_2, we evaluate e_1 and e_2 to integer literals i_1 and i_2 and return `true` if i_1 *op* i_2 holds and `false` otherwise.
- Equality $e_1 = e_2$ evaluates to `true` if the values of e_1 and e_2 are the same, otherwise to `false`. Inequality $e_1 <> e_2$ yields the opposite result.

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!

SOLUTION:

A. State

The state of the compiler consists of the following components:

1. A finite map `context` from identifiers to natural numbers (local variable addresses).
2. A natural number `nextAddress` denoting the next free slot in the JVM variable store of the currently compiled method.
3. A natural number `resultAddress` denoting the address (if any) of the result variable of a function.
4. A stream `labels` of so far unused label names. Elements are taken from this stream whenever a new label name is needed.

Since we are directly printing the generated Jasmin to the standard output, we need not store any generated code. Further, since we pretend that we can use Blaise function identifiers for the corresponding Jasmin methods, we need no function “signature”.

B. Initialization and run

Given a type-correct Blaise program, compilation proceeds as follows:

1. The component `labels` is initialized to an infinite stream of label names.
2. Each function definition is `compile`d (see below)
3. Output: `.method main`
4. The `context` is reset to an empty map and `nextAddress` to 0.
5. Each variable declaration is `compile`d (see below).
6. The main block is `compile`d (see below).
7. Output: `.end method`

C. Compiler

The compiler is specified as an overloaded procedure `compile` in pseudo-code acting on Blaise abstract syntax from Question 1.

```
-- Definitions

compile (FunDf f pars t body):
  emit (.method f)
  context     <- empty
  nextAddress <- 0
  for (x:t in pars): addVar x t
  resultAddress <- addVar f t
  compile body
  emit (iload resultAddress)
  emit (ireturn)
  emit (.end method)

compile (Bdy vars block):
  for (x:t in vars):
    a <- addVar x t
    emit (ldc 0)
    emit (istore a)    -- initialize variable
  compile block

-- Statements

compile (Blck ss):
  for (s : ss) compile s

compile (SBlock block):
  compile block
compile (SEmpty):      -- emit nothing

compile (SAssign x e):
  a <- lookupVar x
  compile e
  emit (istore a)

compile (SWriteln e):
  compile e
  emit (invokestatic writeln)

compile (SIfElse e s1 s2):
  else, done <- newLabel
  compile e
  emit (ifeq else)
  compile s1
  emit (goto done)
  emit (else:)
  compile s2
  emit (done:)

compile (SWhile e s):
  start, done <- newLabel
  emit (start:)
  compile e
  emit (ifeq done)
  compile s
  emit (goto start)
  emit (done:)

compile (SFor x e1 e2 s):
  start, done <- newLabel
  compile e1
  compile e2
  emit (start:)
  emit (dup2)            -- copy bounds for comparison
  emit (if_icmpgt done)
  a <- lookupVar x
  emit (istore a)
  compile s
  emit (iload a)
  emit (ldc 1)
  emit (iadd)            -- bounds are again on top of stack
  emit (goto start)
  emit (done:)
  emit pop2
```

```
-- Expressions

compile (EInt i):
  emit (ldc i)

compile (ETrue):
  emit (ldc 0)

compile (EFalse):
  emit (ldc 1)

compile (EId x):
  a <- lookupVar x
  emit (iload a)

compile (EAdd e1 OMinus e2):
  compile e1
  compile e2
  emit (isub)

compile (ECmp e1 OLt e2):
  true, done <- newLabel
  compile e1
  compile e2
  emit (if_icmplt true)
  emit (ldc 0)
  emit (goto done)
  emit (true:)
  emit (ldc 1)
  emit (done:)

compile (EMul e1 OAnd e2):
  false, done <- newLabel
  compile e1
  emit (dup)
  emit (ifeq done)  -- short-circuit if e1 is false
  emit (pop)
  compile e2
  emit (done:)
```

D. API

- `emit` (*text*): Write *text* and newline to standard output.
- `addVar` x: Set address of x to `nextAddress` in `context`. Increase `nextAddress` by one. Return address of x.
- `lookupVar` x: Return address of x in map `context`.
- `newLabel`: Extract the next element from stream `labels`.