

Programming Language Technology

Exam, 7 April 2020 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 by email (<mailto:andreas.abel@gu.se>, subject: **PLT reexam**) or telephone (+46 31 772 1731).

Exam review: Contact examiner Andreas Abel if you wish an exam review.

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.
- 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. 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:
 - `SSM.cf`
 - `abs.ssm`
 - `Question2.{txt|md|pdf|...}`
 - `Question3.{txt|md|pdf|...}`
 - (other relevant files)

Question 1 (Grammars) Consider the following fantasy assembly language, called *simple stack machine* (SSM). It is a simplification of the Java Virtual Machine. An example SSM program is given on the next page.

- Program: a non-empty list of whitespace-separated *blocks*.
Execution starts with the first block and ends at the end of the last block (no **return** instruction).
- Block: a label followed by a colon and a possibly whitespace-separated list of instructions.
- Label: an identifier starting with a letter and then followed by a possibly empty sequence consisting of letters, digits, and underscores. (Note: this is different from BNFC's **Ident** token type.)
- Instruction: A keyword followed by 0, 1, or 2 whitespace-separated arguments. The following instructions are supported:
 - Unconditional jump: **goto** with 1 argument: label.
 - Conditional jump: **if** with 2 arguments: condition, label.
If the condition holds on the top value of the stack, the jump is executed. Different to the JVM, the stack is *not changed*.
 - Load local variable onto stack: **load** with 1 argument: address.
 - Store top of stack into local variable: **store** with 1 argument: address.
 - Load fixed number onto stack: **const** with 1 argument: literal.
 - Arithmetic operation applied to top two stack elements: one of **add**, **sub**, **mul**, **div** without argument.
As for the JVM, the second operand resides on top of the stack and the first operand is next-to-top. Both operands are removed by the operation and the result is put on the stack instead.
- Condition: one of **pos** (value is positive), **neg** (value is negative), **zero** (value is 0).
- Address: a non-negative decimal number.
- Literal: a non-negative decimal number.

Lines starting with two dashes are comments.

1. Write an SSM program **abs.ssm** that computes the absolute value of a given integer. The input shall reside in local variable 1 and the output shall be left as only entry on the stack.
2. Write a labelled BNF grammar for SSM in a file **SSM.cf** and create a parser from this grammar using BNFC. The parser should be free of conflicts (shift/reduce and reduce/reduce).
3. Test your parser on **abs.ssm** and **factorial.ssm**.

Deliverables: files **SSM.cf** and **abs.ssm**.

```

-- Compute factorial on the simple stack machine (SSM).
--
-- Input n is in local variable 1.
-- Output n! is left on the stack.
-- If input is <= 1, output is 1.

-- The bottom of the stack holds the current result.

begin:  const  1          -- Put 1 (result) onto the stack

-- Put n on the stack and make sure n is positive.
-- From then on, n will never become negative.

        load   1          -- Put n on the stack.
        if neg end        -- If n was negative, nothing to do.

-- Main loop: If top of the stack n == 0, we are done.

loop:   if zero end      -- If this is = 0, goto end.

-- Multiply it onto the result.

        mul

-- Decrement n by one.

        load   1
        const  1
        sub
        store  1

-- Continue with n-1.

        load   1
        goto   loop

-- The product is on the stack, after we remove the final n
-- by storing it in dummy local variable 0.

end:    store  0

```

Question 2 (Interpretation): Write a specification of an interpreter for the simple stack machine (SSM) of Question 1. Input to the interpreter are:

- i. An abstract syntax tree of a SSM program.
- ii. A list of integers serving as the inputs to the SSM program. These, say n , inputs are the starting values for the local variables $1..n$.

The output of the interpreter, if successful, are the values left on the stack after reaching the end on the SSM program. If the program cannot be interpreted, the interpreter shall flag an error instead.

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 you use for each component.
- B. Initialization, run, finalization: Describe how the state is initialized and how the interpreter (C) is started (i.e., which arguments are given to the interpreter). Describe how the output is obtained after the interpreter has finished.
- C. Interpretation: Describe the interpreter: Write an explanation for each case of SSM instruction. You may use pseudo-code if you wish.
- 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 SMM nor access to the course material.

The specification will be judged on clarity and correctness.

Question 3 (Compilation): Consider the following subset **CMM** of the C programming language. (BNFC grammar on following page.)

- Program: a single function definition
- Definition: type followed by function name, parenthesized comma-separated list of declarations followed by a function definition block in braces
- Function definition block: a semicolon-separated list of declarations followed by a list of statements
- Declaration: a type followed by an identifier
- Statement:
 - assignment: identifier followed by =, expression, and semicolon
 - **return** followed by an expression and a semicolon
 - loop: **while** followed by a parenthesized condition and a statement
 - conditional: **if** followed by a parenthesized condition, a statement, **else**, and another statement
 - block: a sequence of statements enclosed in braces
- Condition: $e_1 \text{ op } e_2$ with op one of less-than (<), less-or-equal-than (<=), greater-than (>), greater-or-equal-than (>=), equal (==), unequal (!=)
- Expression e :
 - integer literal
 - identifier
 - arithmetical operation: $e_1 \text{ op } e_2$ with op one of addition (+), subtraction (-), multiplication (*), division (/)
- Type: **int**
- Identifier: letter followed by letters, digits, and underscores

Example: The following **CMM** program computes the factorial of the given argument:

```
/* Factorial function in CMM */

int factorial (int n) {
    int result;
    result = 1;
    while (n > 1) {
        result = result * n;
        n = n - 1;
    }
    return result;
}
```

Specify a compiler from CMM to SSM. The compiler takes an abstract syntax tree of a CMM program as input and translates this into an abstract syntax tree of a SSM program, or throws an exception if the CMM input is ill-formed, e.g., has unbound identifiers.

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, run, finalization, C. Compilation schemes, D. API.

Restriction of the task: From the arithmetical operations, show compilation of *one* form of your choice (+, -, *, or /). Same for the comparison operations. Choose *one* of **if** or **while**.

```

-- BNFC grammar for CMM fragment of C

-- # A Program is a single function definition.
Def.      Program ::= Type Id "(" [Arg] ")" "{" [Decl] [Stm] "}";

-- ## Function parameters.
ADecl.    Arg      ::= Type Id ;
separator Arg    ", " ;

-- ## Variable declarations.
VDecl.    Decl     ::= Type Id ;
terminator Decl  ";" ;

-- # Statements.

SAssign.  Stm      ::= Id "=" Exp ";" ;
SReturn.  Stm      ::= "return" Exp ";" ;
SWhile.   Stm      ::= "while" "(" Cmp ")" Stm ;
SIfElse.  Stm      ::= "if" "(" Cmp ")" Stm "else" Stm ;
SBlock.   Stm      ::= "{" [Stm] "}";
terminator Stm  "" ;

-- # Expressions.

EInt.     Exp2     ::= Integer ;
EId.      Exp2     ::= Id      ;
EMul.     Exp1     ::= Exp1 MulOp Exp2 ; -- Left assoc.
EAdd.     Exp      ::= Exp AddOp Exp1 ; -- Left assoc.
coercions Exp 2 ;

-- # Operators.

OTimes.   MulOp    ::= "*" ;           OPlus.   AddOp    ::= "+" ;
ODiv.     MulOp    ::= "/" ;           OMinus.  AddOp    ::= "-" ;

-- # Comparison.

CCmp.     Cmp      ::= Exp CmpOp Exp ;
OLt.      CmpOp    ::= "<" ;           OGt.     CmpOp    ::= ">" ;
OLtEq.    CmpOp    ::= "<=" ;        OGtEq.   CmpOp    ::= ">=" ;
OEq.      CmpOp    ::= "==" ;         ONEq.    CmpOp    ::= "!=" ;

-- # Types, identifiers, comments

TInt.     Type     ::= "int" ;
token     Id       letter (letter | digit | '_' ) * ;

comment   "/*" "*/" ;

```