# Programming Language Technology

Exam, 14 January 2019 at 8.30–12.30 in M

Course codes: Chalmers DAT151, GU DIT231. As re-exam, also DAT150 and DIT230.
Exam supervision: Daniel Schoepe (+46 31 772 6166), visits at 9:30 and 11:30.

**Grading scale**: Max = 60p, VG = 5 = 48p, 4 = 36p, G = 3 = 24p.
**Allowed aid**: an English dictionary.
**Exam review**: 1 February 2019 10-11.30 in room EDIT 6128.

Please answer the questions in English.

**Question 1 (Grammars):** Write a labelled BNF grammar that covers the following kinds of constructs of C (also C++, Java):

- Programs: A list of statements enclosed between "`int main() {`" and "`}`".
- Statements:
  - statements formed from expressions by adding a semicolon `;`
  - single variable declarations, e.g., `int x;`
  - `while` loops
- Expressions:
  - integer literals
  - identifiers
  - function calls, i.e., identifiers applied to a tuple of expressions
  - addition (`+`) and multiplication (`*`)
  - less-than comparison of integer expressions (`<`)
  - assignments
  - parenthesized expressions

  Function calls, multiplication and addition are left-associative, comparison is non-associative, assignment is right-associative. Function calls bind strongest, then multiplication, then addition, then comparison, then assignment.
- Types: `int` and `bool`

Lines starting with `#` are comments. An example program is:

```
#include <stdio.h>
#define printInt(e) printf("%d\n",e)
int main () {
  int i; int n;
  i = n = 0;
  while (i < 5) n = n + (i = i + 1) * i;
  printInt(n);
}
```

You can use the standard BNFC categories `Integer` and `Ident`, list categories via the `terminator` and `separator` pragmas, and the `coercions` pragma.
(10p)

**CLARIFICATION:** Clarification: blocks are not explicitely mentioned in the list of constructs to be covered, thus, they are not included in this grammar. However, it is not considered a mistake if a solution includes them.

**SOLUTION:**

```
Program.  Prg  ::= "int" "main" "(" ")" "{" Stms "}" ;

TInt.     Type ::= "int"                             ;
TBool.    Type ::= "bool"                            ;

SNil.     Stms ::=                                   ;
SCons.    Stms ::= Stm Stms                          ;

SDecl.    Stm  ::= Type Ident ";"                    ;
SExp.     Stm  ::= Exp ";"                           ;
SWhile.   Stm  ::= "while" "(" Exp ")" Stm           ;

EInt.     Exp3  ::= Integer                          ;
EId.      Exp3  ::= Ident                            ;
ECall.    Exp3  ::= Ident "(" [Exp] ")"              ;
ETimes.   Exp2  ::= Exp2 "*" Exp3                    ;
EPlus.    Exp1  ::= Exp1 "+" Exp2                    ;
ELt.      Exp   ::= Exp1 "<" Exp1                    ;
EAssign.  Exp   ::= Ident "=" Exp                    ;

coercions Exp 3                                      ;
separator Exp ","                                    ;

comment "#"                                          ;
```

**Question 2 (Lexing):** Consider the alphabet $\Sigma = \{0, 1\}$ and the language $L \subseteq \Sigma^*$ of words in which the number of consecutive 1s is always even.

1. Give a regular expression for language $L$.
2. Give a deterministic finite automaton for $L$ with no more than 6 states.

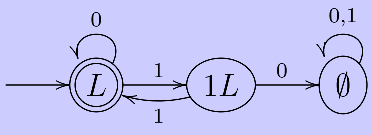
(4p)

**CLARIFICATION:** The *number of consecutive 1s is always even* could be read to allow a single 1, since then there are *no consecutive* 1s, meaning 0 of them, which is an even number. It would habe been clearer to avoid Latin and write *the number of 1s following each other without a 0 inbetween.*

Any of the two interpretations will be accepted as solution.

**SOLUTION:**

1. RE: $(0^*(11)^*)^*$
2. DFA:



**Question 3 (LR Parsing):** Consider the following labeled BNF-Grammar (written in `bnfc`). The starting non-terminal is `P`.

```
P1.     P ::= P "*" F   ;
P2.     P ::= F         ;

F1.     F ::= F "^" E   ;
F2.     F ::= E         ;

EX.     E ::= "x"       ;
EY.     E ::= "y"       ;
EN.     E ::= "-" E     ;
EP.     E ::= "(" P ")" ;
```

Step by step, trace the shift-reduce parsing of the expression

```
- x ^ - - y * x
```

showing how the stack and the input evolves and which actions are performed. (8p)

**SOLUTION:** The actions are `S` (shift), `R` (reduce with rule(s)), and `Accept`. Stack and input are separated by a dot.

```
          . - x ^ - - y * x  // SS
- x       . ^ - - y * x      // R    EX        :  x → E
- E       . ^ - - y * x      // RR   EN, F2    : -E → E → F
F         . ^ - - y * x      // SSSS
F ^ - - y . * x              // RRR  EY,EN,EN  : --y → --E → -E → E
F ^ E     . * x              // RR   F1, P2    : F ^ E → F → P
P         . * x              // SS
P * x     .                  // RR   EX, F1    : x → E → F
P * F     .                  // R    P1        : P * F → P
P         .                  // Accept
```

**Question 4 (Type checking and evaluation):**

1. Write syntax-directed typing rules for the *expressions* of Question 1. Alternatively, you can write the type-checker in pseudo code or Haskell. In any case, the environment must be made explicit. (7p)

**CLARIFICATION:** A function `lookupVar` can be assumed if its behavior is described.

**SOLUTION:** The type checking judgement $\Gamma \vdash_\Sigma e : t$ for expressions is the least relation closed under the following rules.

$$\frac{}{\Gamma \vdash_\Sigma \texttt{EId}\ x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash_\Sigma \texttt{EInt}\ i : \texttt{int}}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : t_1 \quad \ldots \quad \Gamma \vdash_\Sigma e_n : t_n}{\Gamma \vdash_\Sigma \texttt{ECall}\ f\ (e_1, \ldots, e_n) : t}\ \Sigma(f) = (t_1, \ldots, t_n) \to t$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{int} \qquad \Gamma \vdash_\Sigma e_2 : \texttt{int}}{\Gamma \vdash_\Sigma \texttt{ETimes}\ e_1\ e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \texttt{int} \qquad \Gamma \vdash_\Sigma e_2 : \texttt{int}}{\Gamma \vdash_\Sigma \texttt{EPlus}\ e_1\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \texttt{int} \qquad \Gamma \vdash_\Sigma e_2 : \texttt{int}}{\Gamma \vdash_\Sigma \texttt{ELt}\ e_1\ e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash_\Sigma e : t}{\Gamma \vdash_\Sigma \texttt{EAssign}\ x\ e : t}\ \Gamma(x) = t$$

Herein, $\Gamma$ is a finite map from identifiers $x$ to types $t$, and $\Sigma$ a finite map from identifiers $f$ to function types $(t_1, \ldots, t_n) \to t$.

2. Write syntax-directed interpretation rules for the *statements* of Question 1, assuming an interpreter for expressions $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$. Alternatively, you can write the interpreter in pseudo code or Haskell. In any case, the environment must be made explicit. (5p)

**CLARIFICATION:** A function `lookupVar` can be assumed if its behavior is described.

**SOLUTION:** The evaluation judgement $\gamma \vdash s \Downarrow \gamma'$ for statements is the least relation closed under the following rules.

$$\frac{}{\gamma \vdash \texttt{SDecl}\ t\ x \Downarrow (\gamma, x{=}\mathsf{void})} \qquad \frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash \texttt{SExp}\ e \Downarrow \gamma'} \qquad \frac{\gamma \vdash e \Downarrow \langle \texttt{false}; \gamma' \rangle}{\gamma \vdash \texttt{SWhile}\ e\ s \Downarrow \gamma'}$$

$$\frac{\gamma \vdash e \Downarrow \langle \texttt{true}; \gamma' \rangle \qquad (\gamma'.\varepsilon) \vdash s \Downarrow (\gamma''.\delta) \qquad \gamma'' \vdash \texttt{SWhile}\ e\ s \Downarrow \gamma'''}{\gamma \vdash \texttt{SWhile}\ e\ s \Downarrow \gamma'''}$$

Herein, environment $\gamma$ is a dot-separated list of blocks $\delta$, each of which is a finite map from identifiers $x$ to values $v$. We write $\varepsilon$ for the empty map and $\gamma, x = v$ for extending the top block of $\gamma$ by the binding $x = v$.

**Question 5 (Compilation):**

1. Write compilation schemes in pseudo code or Haskell for the statement and expressions constructions of Question 1. The compiler should output symbolic JVM instructions (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. (9p)

**CLARIFICATION:** A function `lookupVar` can be assumed if its behavior is described.

**SOLUTION:**

```
-- Compilation of expressions

compile (EInt i) = do
  emit (ldc i)

compile (EId x) = do
  a <- lookupVar x      -- variable x has address a in variable store
  emit (iload a)

compile (ECall f es) = do
  m <- lookupFun f      -- get the Jasmin name of function f
  mapM_ compile es
  emit (invokestatic m)

compile (EAssign x e) = do
  a <- lookupVar x      -- variable x has address a in variable store
  compile e
  emit (istore a)
  emit (iload a)

compile (ETimes e e') = do
  compile e
  compile e'
  emit (imul)

compile (EPlus e e') = do
  compile e
  compile e'
  emit (iadd)

compile (ELt e e') = do
  done <- newLabel
  emit (ldc 1)          -- speculate that e < e' holds
  compile e
  compile e'
  emit (if_icmplt done) -- test e < e'
  emit (pop)            -- test failed, replace 1 by 0
  emit (ldc 0)
  emit (done:)

-- Compilation of statements

compile (SDecl t x) = do
  addVar t x            -- register local variable x, emit no code

compile (SExp e) = do
  compile e
  emit (pop)

compile (SWhile e s) = do
  start, done <- newLabel
  emit (start:)
  compile e             -- condition
  emit (ifeq done)      -- if false, exit loop
  compile s
  emit (goto start)     -- rerun loop
  emit (done:)
```

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 $(P, V, S)$ is 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. (7p)

**SOLUTION:** Stack $S.v$ shall mean that the top value on the stack is $v$, the rest is $S$. Jump targets $L$ are used as instruction addresses, and $P + 1$ is the instruction address following $P$.

| instruction | state before | | state after | |
|---|---|---|---|---|
| `goto` $L$ | $(P, V, S)$ | $\rightarrow$ | $(L, V, S)$ | |
| `ifeq` $L$ | $(P, V, S.0)$ | $\rightarrow$ | $(L, V, S)$ | |
| `ifeq` $L$ | $(P, V, S.v)$ | $\rightarrow$ | $(P + 1, V, S)$ | if $v \neq 0$ |
| `if_icmplt` $L$ | $(P, V, S.v.w)$ | $\rightarrow$ | $(L, V, S)$ | if $v < w$ |
| `if_icmplt` $L$ | $(P, V, S.v.w)$ | $\rightarrow$ | $(P + 1, V, S)$ | unless $v < w$ |
| `iload` $a$ | $(P, V, S)$ | $\rightarrow$ | $(P + 1, V, S.V(a))$ | |
| `istore` $a$ | $(P, V, S.v)$ | $\rightarrow$ | $(P + 1, V[a := v], S)$ | |
| `ldc` $i$ | $(P, V, S)$ | $\rightarrow$ | $(P + 1, V, S.i)$ | |
| `imul` | $(P, V, S.v.w)$ | $\rightarrow$ | $(P + 1, V, S.(v \cdot w))$ | |
| `iadd` | $(P, V, S.v.w)$ | $\rightarrow$ | $(P + 1, V, S.(v + w))$ | |
| `pop` | $(P, V, S.v)$ | $\rightarrow$ | $(P + 1, V, S)$ | |
| `invokestatic` $m$ | $(P, V, S.v_1 \ldots v_n)$ | $\rightarrow$ | $(P + 1, V, S.v)$ where $v = m(v_1, \ldots, v_n)$ | |

**Question 6 (Functional languages):**

1. For the expressions of lambda-calculus with integers and integer addition we use the grammar

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

and for simple types $t ::= \mathbb{N} \mid t \to t$. Non-terminal $x$ ranges over variable names and $n$ over 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) | $k : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ | $\vdash 1 + k\,(\lambda x \to x)$ | $: \mathbb{N}$ |
| (b) | | $\vdash \lambda y \to \lambda h \to (h\, 1)\, y$ | $: \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ |
| (c) | $x : \mathbb{N} \to \mathbb{N},\ g : \mathbb{N}$ | $\vdash (g + 1)\, x$ | $: \mathbb{N}$ |
| (d) | $f : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ | $\vdash (\lambda g \to f\, g)\, (\lambda y \to f\, (\lambda z \to z)\, y)$ | $: \mathbb{N} \to \mathbb{N}$ |
| (e) | $f : \mathbb{N} \to \mathbb{N}$ | $\vdash \lambda x \to f\, (1 + f\, (f\, x))$ | $: \mathbb{N} \to \mathbb{N}$ |

*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)

**CLARIFICATION:** The original problem (d) had a spurious parenthesis at the end: (d) $f : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \vdash (\lambda g \to f\, g)\, (\lambda y \to f\, (\lambda z \to z)\, y)) : \mathbb{N} \to \mathbb{N}$

**SOLUTION:**

(a) valid

(b) not valid ($h$ is not a function, cannot apply it to 1)

(c) not valid ($g + 1$ is not a function)

(d) valid

(e) valid

2. Write a **call-by-value** interpreter for above lambda-calculus either with inference rules, or in pseudo code or Haskell. (5p)

**SOLUTION:**

```
type Var = String
data Exp = EInt Integer | EPlus Exp Exp
         | EVar Var | EAbs Var Exp | EApp Exp Exp

data Val = VInt Integer | VClos Var Exp Env
type Env = [(Var,Val)]

eval :: Exp -> Env -> Maybe Val
eval e0 rho = case e0 of
  EInt n   -> return (VInt n)
  EAbs x e -> return (VClos x e rho)
  EVar x   -> lookup x rho

  EApp f e -> do
    VClos x e' rho' <- eval f rho
    v               <- eval e rho
    eval e' ((x,v) : rho')

  EPlus e1 e2 -> do
    VInt i1 <- eval e1 rho
    VInt i2 <- eval e2 rho
    return (VInt (i1 + i2))
```