Programming Language Technology

Exam, 11 January 2017 at 8.30–12.30 in J

Course codes: Chalmers DAT150/151, GU DIT231. As re-exam, also TIN321 and DIT229/230.
Exam supervision: Fredrik Lindblad (+46 31 772 2038), visits at 9:30 and 11:30.
[Examiner: Andreas Abel (+49 176 400 333 23)]

Grading scale: Max = 60p, VG = 5 = 48p, 4 = 36p, G = 3 = 24p.
Allowed aid: an English dictionary.
Exam review: Tuesday 24 January 2017 at 10-12 in room EDIT 5128.

Please answer the questions in English. Questions requiring answers in code can be answered in any of: C, C++, Haskell, Java, or precise pseudocode.

For any of the six questions, an answer of roughly one page should be enough.

Question 1 (Grammars): Write a BNF grammar that covers the following kinds of constructs in Java/C++:

- statements:
  - blocks: lists of statements (possibly empty) in curly brackets { }
  - variable initialization statement: a type followed by an identifier and an initializing expression, e.g. `int x = 4`

    ERRATUM: Missing here: *followed by a semicolon.*
    Example should be `int x = 4;`

  - expressions as statements: an expression followed by a semicolon
- types: `int`
- expressions:
  - variables
  - integer literals
  - addition `+`
  - multiplication `*`
  - assignment to variables, e.g. `x = (y = 3) + z`

  Both arithmetic operations are left associative. It is enough to consider 4 precedence levels of expressions (from lowest to highest): assignment (right associative), addition, multiplication, and atoms. Parentheses are used, as usual, to lift an expression to the highest level.

An example statement is shown in question 2. You can use the standard BNFC categories `Integer` and `Ident`, as well as `coercions`. **Do not** use list categories or `terminator/separator` rules. (10p)

SOLUTION:

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

SBlock. Stm ::= "{" Stms "}" ;
SInit.  Stm ::= Type Ident "=" Exp ";" ;
SExp.   Stm ::= Exp ";" ;

TInt.  Type ::= "int" ;

EVar. Exp3 ::= Ident ;
EInt. Exp3 ::= Integer ;
EMul. Exp2 ::= Exp2  "*" Exp3 ;
EAdd. Exp1 ::= Exp1  "+" Exp2 ;
EAss. Exp  ::= Ident "=" Exp ;
coercions Exp 3;
```

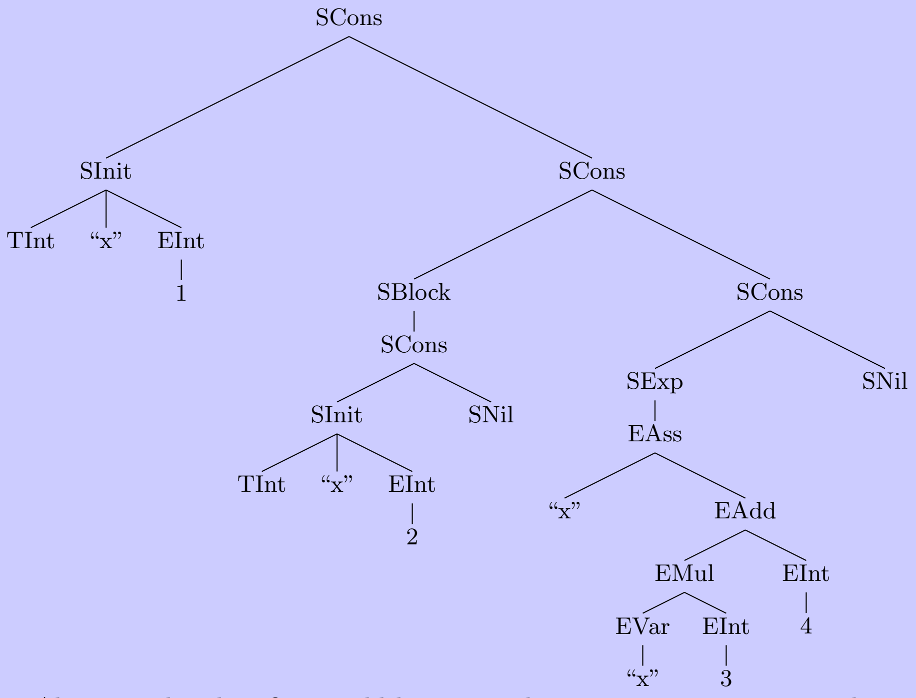
Question 2 (Trees): Show the parse tree and the abstract syntax tree of the statement

ERRATUM: Should be *list of statements*

```
int x = 1; { int x = 2; } x = x * 3 + 4;
```

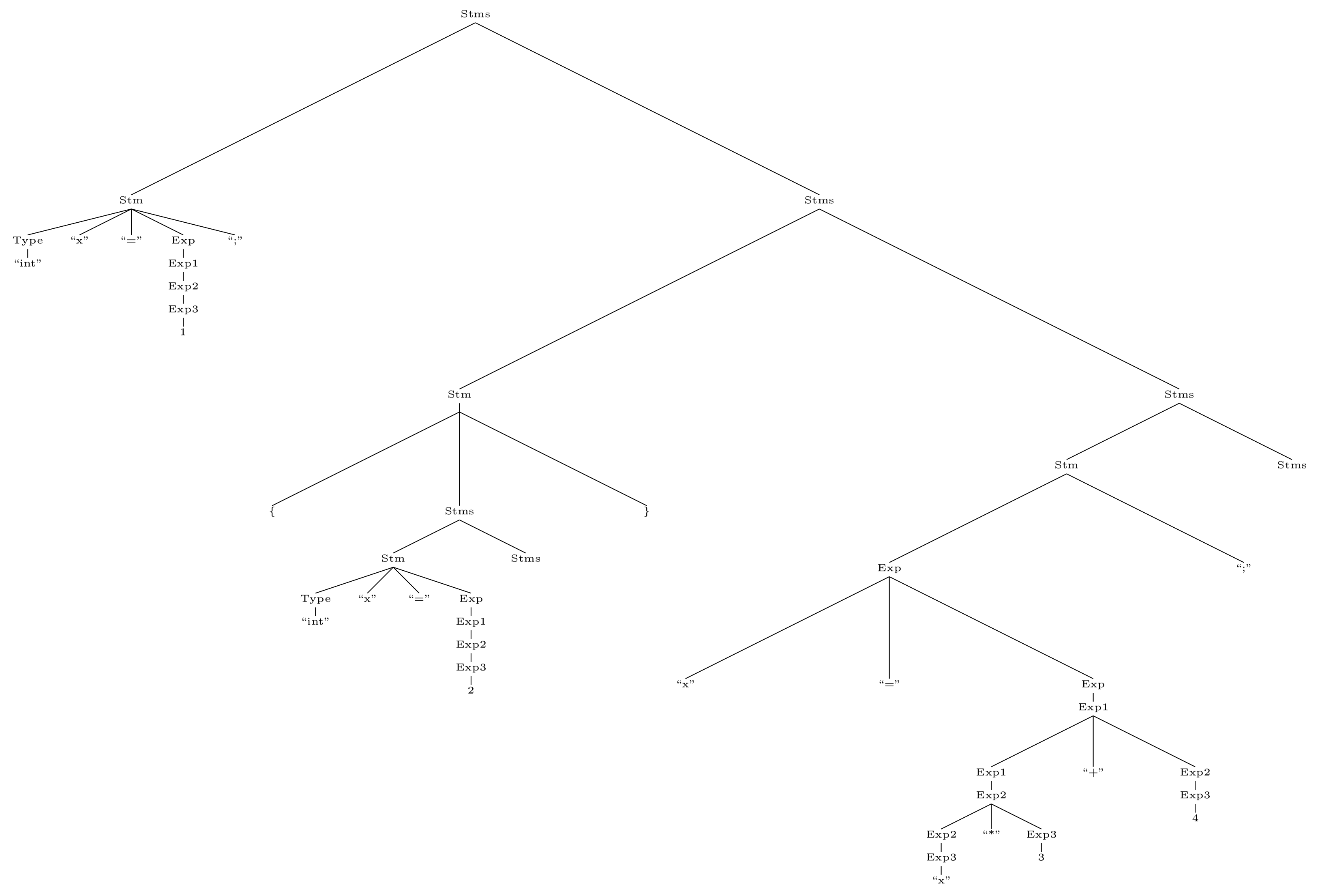
in the grammar that you wrote in Question 1. Note: `Ident` and `Integer` wrappers for identifier and integer tokens should be either always supplied, or always dropped; be consistent! (10p)

SOLUTION:
The **abstract syntax tree** is:



Alternatively, identifiers could be wrapped in an `Ident` constructor; this is what `bnfc` does. Note that this constructor would not come from the parser, but from the lexer.

The **parse tree** should not use rule names, but non-terminals and terminals only. Also, it needs to explicit about coercions. (Parse tree omitted here due to lack of artistic ambition, see similar question in old exams.)


Stms
Stm
Stms
Type
“int”
“x”
“=”
Exp
Exp1
Exp2
Exp3
1
“;”
Stm
Stms
{
Stms
}
Stm
Stms
Type
“int”
“x”
“=”
Exp
Exp1
Exp2
Exp3
2
Stm
Stms
Exp
“;”
“x”
“=”
Exp
Exp1
Exp1
“+”
Exp2
Exp2
Exp3
4
Exp2
“*”
Exp3
Exp3
3
“x”

Question 3 (Type checking and evaluation):

1. Write syntax-directed *type checking* rules for the *statement* forms and lists of Question 1. The typing environment must be made explicit. You can assume a type-checking judgement for expressions. (5p)

SOLUTION: We use a judgement $\Gamma \vdash s \Rightarrow \Gamma'$ that expresses that statement s is well-formed in context Γ and might introduce new declarations, resulting in context Γ'.

A context Γ is a stack of blocks Δ, separated by a dot. Each block Δ is a map from variables x to types t. We write $\Delta, x{:}t$ for adding the binding $x \mapsto t$ to the map. Duplicate declarations of the same variable in the same block are forbidden; with $x \notin \Delta$ we express that x is not bound in block Δ. We use a judgement $\Gamma \vdash e : t$, which reads "in context Γ, expression e has type t".

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{SExp}\, e \Rightarrow \Gamma} \qquad \frac{\Gamma. \vdash ss \Rightarrow \Gamma.\Delta}{\Gamma \vdash \texttt{SBlock}\, ss \Rightarrow \Gamma}$$

$$\frac{\Gamma.\Delta \vdash e : t}{\Gamma.\Delta \vdash \texttt{SInit}\, t\, x\, e \Rightarrow (\Gamma.\Delta, x{:}t)}\ x \notin \Delta$$

This judgement is extended to sequences of statements $\Gamma \vdash ss \Rightarrow \Gamma'$ by the following rules:

$$\frac{}{\Gamma \vdash \texttt{SNil} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash s \Rightarrow \Gamma' \qquad \Gamma' \vdash ss \Rightarrow \Gamma''}{\Gamma \vdash \texttt{SCons}\, s\, ss \Rightarrow \Gamma''}$$

Alternative solution: Lists of statements are denoted by ss and ε is the empty list. The judgement $\Gamma \vdash ss$ reads "in context Γ, the sequence of statements ss is well-formed". Here, concrete syntax is used for the statements:

$$\frac{}{\Gamma \vdash \varepsilon} \qquad \frac{\Gamma. \vdash ss \qquad \Gamma \vdash ss'}{\Gamma \vdash \{ss\}ss'}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash ss}{\Gamma \vdash e; ss} \qquad \frac{\Gamma.\Delta \vdash e : t \qquad \Gamma.\Delta, x : t \vdash ss}{\Gamma.\Delta \vdash t\, x = e; ss}\ x \notin \Delta$$

2. Write syntax-directed *interpretation* rules for the *expression* forms of Question 1. The environment must be made explicit, as well as all possible side effects. (5p)

SOLUTION:

The judgement $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$ reads "in environment γ, evaluation of the expression e results in value v and environment γ'".

$$\frac{}{\gamma \vdash \texttt{EInt}\, i \Downarrow \langle i; \gamma \rangle} \qquad \frac{}{\gamma \vdash \texttt{EVar}\, x \Downarrow \langle \gamma(x); \gamma \rangle}$$

$$\frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma_1 \rangle \qquad \gamma_1 \vdash e_2 \Downarrow \langle i_2; \gamma_2 \rangle}{\gamma \vdash \texttt{EAdd}\, e_1\, e_2 \Downarrow \langle i_1 + i_2; \gamma_2 \rangle}$$

$$\frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma_1 \rangle \qquad \gamma_1 \vdash e_2 \Downarrow \langle i_2; \gamma_2 \rangle}{\gamma \vdash \texttt{EMul}\, e_1\, e_2 \Downarrow \langle i_1 * i_2; \gamma_2 \rangle} \qquad \frac{\gamma \vdash e \Downarrow \langle v; \gamma' \rangle}{\gamma \vdash \texttt{EAss}\, x\, e \Downarrow \langle v; \gamma'[x := v] \rangle}$$

Question 4 (Parsing): Step by step, trace the LR-parsing of the expression

```
x = x * 3 + 4
```

showing how the stack and the input evolves and which actions are performed. Be careful that the actions match your grammar in Question 1. (8p)

SOLUTION: `x` is actually token `Ident`. `1`, `3`, `4` are token `Integer`, abbreviated to `Int`. Actions are `S` (shift), `R` (reduce with rule(s)), and `A` (accept).

```
                    . x = x * 3 + 4  // SSS
Ident = Ident       . * 3 + 4        // R: Ident -> Exp3 -> Exp2
Ident = Exp2        . * 3 + 4        // SS
Ident = Exp2 * Int  . + 4            // R: Int -> Exp3
Ident = Exp2 * Exp3 . + 4            // R: Exp2 * Exp3 -> Exp2
Ident = Exp2        . + 4            // R: Exp2 -> Exp1
Ident = Exp1        . + 4            // SS
Ident = Exp1 + Int                   // R: Int -> Exp3 -> Exp2
Ident = Exp1 + Exp2                  // R: Exp1 + Exp2 -> Exp1
Ident = Exp1                         // R: Exp1 -> Exp
Ident = Exp                          // R: Ident = Exp -> Exp
Exp                                  // A
```

Question 5 (Compilation):

1. Write compilation schemes in pseudo-code for each of the grammar constructions in Question 1 generating JVM (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. (6p)

SOLUTION:

```
// Blocks

compile (SBlock ss) = do
  newBlock
  mapM_ compile ss
  popBlock

compile (SInit t x e) = do
  newVar t x
  a <- lookupVar x
  compile e
  emit (istore a)

compile (SExp e) = do
  compile e
  emit pop

compile (EVar x) = do
  a <- lookupVar x
  emit (iload a)

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

compile (EAdd e1 e2) = do
  compile e1
  compile e2
  emit (iadd)

compile (EMul e1 e2) = do
  compile e1
  compile e2
  emit (imul)

compile (EAss x e) = do
  compile e
```

```
a <- lookupVar x
istore a
iload a
```

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. (6p)

SOLUTION:

$$
\begin{array}{llllll}
\texttt{ldc a} & : & (P, V, S) & \longrightarrow & (P+1, V, & S.a) \\
\texttt{iload x} & : & (P, V, S) & \longrightarrow & (P+1, V, & S.V(x)) \\
\texttt{istore x} & : & (P, V, S.a) & \longrightarrow & (P+1, V[x=a], S) & \\
\texttt{pop} & : & (P, V, S.a) & \longrightarrow & (P+1, V, & S) \\
\texttt{iadd} & : & (P, V, S.a.b) & \longrightarrow & (P+1, V, & S.(a+b)) \\
\texttt{imul} & : & (P, V, S.a.b) & \longrightarrow & (P+1, V, & S.(a*b))
\end{array}
$$

Question 6 (Functional languages):

1. Give the typing rules for simply-typed lambda-calculus!
   Simple types are given by the grammar $t ::= \texttt{int} \mid t \to t$, and expressions by $e ::= x \mid \lambda x \to e \mid e\,e$. (5p)

SOLUTION:

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x \to e : t \to t'} \qquad \frac{\Gamma \vdash e : t' \to t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash e\,e' : t}$$

2. Give a typing derivation of $\lambda g \to \lambda f \to f\,(g\,f)$. (5p)

SOLUTION:

$$\begin{aligned}
&\Gamma := (g : (t \to t') \to t,\ \ f : t \to t') \\
&\Gamma \vdash g\,f : t \\
&\Gamma \vdash f\,(g\,f) : t' \\
&\vdash \lambda g \to \lambda f \to f\,(g\,f) : ((t \to t') \to t) \to (t \to t') \to t'
\end{aligned}$$