Programming Language Technology

Exam, 11 January 2024, 8.30–12.30 at Johanneberg Campus

Course codes: Chalmers DAT151, GU DIT231.
Exam supervision: Andreas Abel (+46 31 772 1731), visits at 9:30 and 11:30.

Grading scale: Max = 60p, MVG = 5 = 48p, VG = 4 = 36p, G = 3 = 24p.
Allowed aid: an English dictionary.
Exam review: Thu 18 January 2024 14.30-15.30 in room EDIT 3128.

Please answer the questions in English.

Question 1 (Grammars): Write a labelled BNF grammar that covers the following kinds of constructs of C:

- Program: `int main()` followed by a block
- Block: a sequence of statements enclosed between `{` and `}`
- Statement:
  - statement formed from an expression by adding a semicolon `;`
  - initializing variable declarations, e.g., `int x = e;`
  - assignment, e.g., `x = e;`
  - loop: `while` followed by a parenthesized expression and a block
- Atomic expression:
  - identifier
  - integer literal
  - function call with a single argument
  - pre-increment of identifier, e.g., `++x`
  - parenthesized expression
- Expression (from highest to lowest precedence):
  - atomic expression
  - addition (`+`), left-assocative
  - less-than comparison of integer expressions (`<`), non-associative
- Type: `int` or `bool`

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

```
#include <stdio.h>
#define printInt(i) printf("%d\n",i)
int main ()
{ int n = 42;  int i = 0;  int k = 0;
  while (k < 101) { n = k;  k = n + ++i; }
  printInt(n);
}
```

You can use the standard BNFC categories `Integer` and `Ident` and the `coercions` pragma. Do **not** use list categories via the `terminator` and `separator` pragmas!
(10p)

SOLUTION:

```
Program.  Prg   ::= "int" "main" "(" ")" Block  ;

SBlock.   Block ::= "{" Stms "}"                ;

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

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

EInt.     Exp2  ::= Integer                     ;
EId.      Exp2  ::= Ident                       ;
EPreIncr. Exp2  ::= "++" Ident                  ;
ECall.    Exp2  ::= Ident "(" Exp ")"           ;
EPlus.    Exp1  ::= Exp1 "+" Exp2               ;
ELt.      Exp   ::= Exp1 "<" Exp1               ;

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

coercions Exp 2                                 ;

comment "#"                                     ;
```

Question 2 (Lexing): An *non-nested C block comment* starts with `/*` and ends with `*/` and can have any characters in between (but not the comment-end sequence `*/` of course). Also, `/*/` is *not* a valid comment.

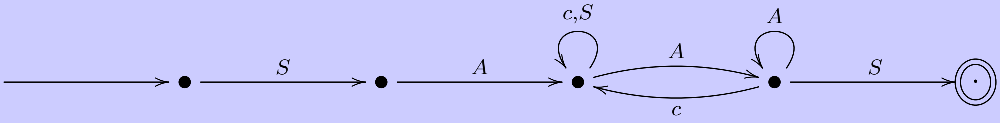
1. Give a deterministic finite automaton for such comments with no more than 8 states. Remember to mark initial and final states appropriately.

2. Give a regular expression for such comments.

Work in the alphabet $\{S, A, c\}$ distinguishing 3 tokens: S for `'/'`, A for `'*'`, and c where c stands for *any other character.* (6p)

SOLUTION:

1. DFA:



2. RE: E.g. $SA\big(A^*c \mid S\big)^* AA^*S$

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

```
Start. S ::= M P     ;

MEmp.  M ::=         ;
MBin.  M ::= M A "*" ;

PEmp.  P ::=         ;
PBin.  P ::= A "+" P ;

X.     A ::= "x"     ;
Y.     A ::= "y"     ;
```

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

```
x * y * y + x +
```

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

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

```
            . x * y * y + x + -- reduce with rule MEmp
M           . x * y * y + x + -- shift
M x         . * y * y + x +   -- reduce with rule X
M A         . * y * y + x +   -- shift
M A *       . y * y + x +     -- reduce with rule MBin
M           . y * y + x +     -- shift
M y         . * y + x +       -- reduce with rule Y
M A         . * y + x +       -- shift
M A *       . y + x +         -- reduce with rule MBin
M           . y + x +         -- shift
M y         . + x +           -- reduce with rule Y
M A         . + x +           -- shift
M A +       . x +             -- shift
M A + x     . +               -- reduce with rule X
M A + A     . +               -- shift
M A + A +   .                 -- reduce with rule PEmp
M A + A + P .                 -- reduce with rule PBin
M A + P     .                 -- reduce with rule PBin
M P         .                 -- reduce with rule Start
S           .                 -- accept
```

Question 4 (Type checking and evaluation):

1. Write syntax-directed *type checking* rules for the *statement* forms and blocks of Question 1. Observe the scoping rules for variables! You can assume a type-checking judgement for expressions.

   Alternatively, you can write the type checker in pseudo code or Haskell. In any case, the typing environment must be made explicit. ~~(8p)~~(7p)

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'. Judgement $\Gamma \vdash b$ states that block b is well-formed in 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 refer to a judgement $\Gamma \vdash e : t$, which reads "in context Γ, expression e has type t".

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

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{SExp}\; e \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash b}{\Gamma \vdash \texttt{SWhile}\; e\; b \Rightarrow \Gamma} \qquad \frac{\Gamma. \vdash ss \Rightarrow \Gamma.\Delta}{\Gamma \vdash \texttt{SBlock}\; ss}$$

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

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

2. Write syntax-directed *interpretation* rules for the *expressions* of Question 1. You can leave out function calls.

   Alternatively, you can write the interpreter in pseudo code or Haskell. A function `lookupVar` can be assumed if its behavior is described. In any case, the environment must be made explicit. ~~(6p)~~(5p)

SOLUTION: The evaluation judgement $\gamma \vdash e \Downarrow \langle v; \gamma' \rangle$ for expressions is the least relation closed under the following rules.

$$\frac{}{\gamma \vdash \texttt{EId}\; x \Downarrow \langle \gamma(x); \gamma \rangle} \qquad \frac{}{\gamma \vdash \texttt{EInt}\; i \Downarrow \langle i; \gamma \rangle} \qquad \frac{i = \gamma(x) + 1}{\gamma \vdash \texttt{EPreIncr}\; x \Downarrow \langle i; \gamma[x := i] \rangle}$$

$$\frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle i_2; \gamma'' \rangle}{\gamma \vdash \texttt{EAdd}\; e_1\; e_2 \Downarrow \langle i_1 {+} i_2; \gamma'' \rangle} \qquad \frac{\gamma \vdash e_1 \Downarrow \langle i_1; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle i_2; \gamma'' \rangle}{\gamma \vdash \texttt{ELt}\; e_1\; e_2 \Downarrow \langle i_1 {<} i_2; \gamma'' \rangle}$$

Question 5 (Compilation):

1. *Statement by statement*, translate the function `main` of the example program of Question 1 to Jasmin. (Do not optimize the program before translation!)

   To translate the call to `printInt`, assume a Java class `Runtime` with a method `void printInt(int)`.

   Make clear which instructions come from which statement, and determine the stack and local variable limits. Please remember that JVM methods must end in a return instruction. (7p)

SOLUTION:

```
.method public static main()I
.limit locals 3
.limit stack  2

        ;; int n = 42;
        ldc           42
        istore        0       ;; n

        ;; int i = 0;
        ldc           0
        istore        1       ;; i

        ;; int k = 0;
        ldc           0
        istore        2       ;; k

        ;; while (k < 101)
L0:     iload         2       ;; k
        ldc           101
        if_icmpge     L1

        ;; n = k;
        iload         2       ;; k
        istore        0       ;; n

        ;; k = n + ++ i;
        iload         0       ;; n
        iinc          1 1     ;; i
        iload         1       ;; i
        iadd
        istore        2       ;; k
        goto          L0

        ;; printInt (n);
L1:     iload         0       ;; n
```

```
        invokestatic Runtime/printInt(I)V

        ;; return 0;  // mandatory return from main added by compiler
        ldc          0
        ireturn

.end method
```

2. Give the small-step semantics of the JVM instructions you used in the Jasmin code in part 1 (except for `return` instructions). 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, 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)	$\to$	(L, V, S)	
`if_icmpge` L	$(P, V, S.v.w)$	$\to$	(L, V, S)	if $v \geq w$
`if_icmpge` L	$(P, V, S.v.w)$	$\to$	$(P + 1, V, S)$	unless $v \geq w$
`iload` a	(P, V, S)	$\to$	$(P + 1, V, S.V(a))$	
`istore` a	$(P, V, S.v)$	$\to$	$(P + 1, V[a := v], S)$	
`ldc` i	(P, V, S)	$\to$	$(P + 1, V, S.i)$	
`inc` a i	(P, V, S)	$\to$	$(P + 1, V[a := V(a) + i], S)$	
`iadd`	$(P, V, S.v.w)$	$\to$	$(P + 1, V, S.(v + w))$	
`invokestatic` m	$(P, V, S.v_1 \dots v_n)$	$\to$	$(P + 1, V, S.v)$	where $v = m(v_1, \dots, v_n)$

Question 6 (Functional languages):

1. The following grammar describes a tiny simply-typed sub language of Haskell.

x			identifier
n	$::=$	$0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \dots$	numeral
e	$::=$	$n \mid e + e \mid x \mid \lambda x \to e \mid e\, e$	expression
t	$::=$	$\mathsf{Int} \mid t \to t$	type

Application $e_1\ e_2$ is left-associative, the arrow $t_1 \to t_2$ is right-associative. Application binds strongest, then addition, then λ-abstraction.

For the following typing judgements $\Gamma \vdash e : t$, decide whether they are valid or not. Your answer can be just "valid" or "not valid", but you may also provide a justification why some judgement is invalid.

(a) $x : \mathsf{Int} \to \mathsf{Int},\ g : \mathsf{Int} \vdash x\,(y+1) : \mathsf{Int}$
(b) $h : \mathsf{Int} \to \mathsf{Int} \vdash \lambda y \to \lambda h \to (h+1) + y : \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$
(c) $k : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \vdash k\,(\lambda f \to f) + 1 : \mathsf{Int}$
(d) $x : \mathsf{Int} \to \mathsf{Int} \vdash \lambda f \to f\,(1 + f\,(f\,x)) : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$
(e) $f : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int}) \vdash (\lambda i \to f\,i)\,(\lambda y \to f\,(\lambda h \to h)\,y) : \mathsf{Int} \to \mathsf{Int}$

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)

SOLUTION:

(a) not valid (y is not in scope)
(b) valid
(c) valid
(d) not valid ($f\,x$ is not function, but f expects one)
(e) valid

2. For each of the following terms, decide whether it evaluates more efficiently (in the sense of fewer reductions) in call-by-name or call-by-value. Your answer can be just "call-by-name" or "call-by-value", but you can also add a justification why you think so. *Same rules for multiple choice as in part 1.* (5p)

(a) $(\lambda x \to \lambda y \to x + x)\ (1+2)\ (3+4+5+6)$
(b) $(\lambda x \to \lambda y \to x + x)\ (1+2+3+4)\ (5+6)$
(c) $(\lambda x \to \lambda y \to y + y)\ ((\lambda z \to z\,z)(\lambda z \to z\,z))\ (1+2+3)$
(d) $(\lambda x \to \lambda y \to y + y)\ (\lambda u \to (\lambda z \to z\,z)(\lambda z \to z\,z))\ (1+2+3+4)$
(e) $(\lambda x \to x + x)\ ((\lambda y \to \lambda z \to z + z)\ (1+2+3)\ (4+5+6))$

SOLUTION:

(a) call-by-name (3 additions vs. 5)
(b) call-by-value (5 additions vs. 7)
(c) call-by-name (diverges in call-by-value)
(d) call-by-value (4 additions vs. 7)
(e) call-by-value (6 additions vs. 11)