Programming Language Technology

Exam, 4 April 2024, 8.30–12.30 in SB-L308

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, VG = 5 = 48p, 4 = 36p, G = 3 = 24p.
Allowed aid: an English dictionary.
Exam review: Wed 17 April 2024 9.30-10.30 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:

- Program: `int main()` followed by a block
- Block: a sequence of statements enclosed between `{` and `}`
- Statements:
  - blocks
  - expression followed by semicolon
  - initializing single variable declarations, e.g., `int x = e;`
  - loop: `while` followed by a parenthesized expression and a statement
- Expressions:
  - identifiers
  - integer literals
  - preincrements (`++x`) and postincrements (`x++`) of identifiers (`x`)
  - less-than comparison of integer expressions (`<`)
  - boolean conjunction (`&&`)

  Comparison is non-associative and binds stronger than the left-associative conjunction.
- Types: `int` and `bool`

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

```
#include <stdio.h>
#define printInt(i) printf("%d\n",i)
int main ()
{ int n = 0;  int k = 0;
  while (k++ < 10) { int i = 0;  while (i++ < k) n++; }
  // printInt(n);
}
```

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

SOLUTION:

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

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

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

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

EInt.       Exp2  ::= Integer                     ;
EId.        Exp2  ::= Ident                       ;
EPreIncr.   Exp2  ::= "++" Ident                  ;
EPostIncr.  Exp2  ::= Ident "++"                  ;
ELt.        Exp1  ::= Exp2 "<" Exp2               ;
EAnd.       Exp   ::= Exp "&&" Exp1               ;

coercions   Exp 2                                 ;

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

comment "#"                                       ;
comment "//"                                      ;
```

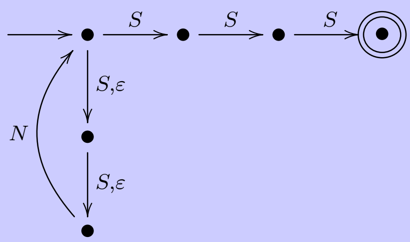
Question 2 (Lexing): You roll a dice until you get a six three times in a row. Let $L \subseteq \Sigma^*$ be the language of such roll sequences. You can work with the alphabet $\Sigma = \{S, N\}$ where S stands for a six and N for a non-six (one to five).

1. Give a regular expression for language L.
2. Give a non-deterministic finite automation for L.
3. Give a *minimal* deterministic finite automaton for L.

(6p)

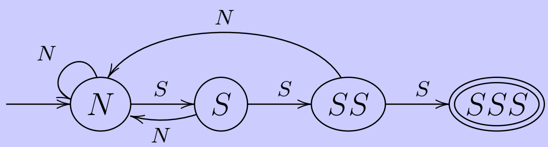
SOLUTION:

1. RE: $(S^? S^? N)^* SSS$ where $S^? = (S \mid \varepsilon)$
2. NFA:



Of course, the following DFA would also be a possible solution for the NFA. (Every DFA is trivially a NFA.)

3. DFA:



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

```
D1.     D ::= D "|" C    ;
D2.     D ::= C          ;

C1.     C ::= C "&" L    ;
C2.     C ::= L          ;

LA.     L ::= "A"        ;
LB.     L ::= "B"        ;
LN.     L ::= "~" L      ;
LP.     L ::= "(" D ")"  ;
```

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

```
~ A & ~ ~ B | A
```

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.

```
          . ~ A & ~ ~ B | A  -- shift
~         . A & ~ ~ B | A    -- shift
~ A       . & ~ ~ B | A      -- reduce with rule LA
~ L       . & ~ ~ B | A      -- reduce with rule LN
L         . & ~ ~ B | A      -- reduce with rule C2
C         . & ~ ~ B | A      -- shift
C &       . ~ ~ B | A        -- shift
C & ~     . ~ B | A          -- shift
C & ~ ~   . B | A            -- shift
C & ~ ~ B . | A              -- reduce with rule LB
C & ~ ~ L . | A              -- reduce with rule LN
C & ~ L   . | A              -- reduce with rule LN
C & L     . | A              -- reduce with rule C1
C         . | A              -- reduce with rule D2
D         . | A              -- shift
D |       . A                -- shift
D | A     .                  -- reduce with rule LA
D | L     .                  -- reduce with rule C2
D | C     .                  -- reduce with rule D1
D         .                  -- halt
```

Question 4 (Type checking and evaluation):

1. Write syntax-directed *type checking* rules for the *expression* forms of Question 1. Alternatively, you can write the type checker in pseudo code or Haskell. (E.g., Java is *not* pseudo code!) In any case, the typing environment must be made explicit. (6p)

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

$$\frac{}{\Gamma \vdash \texttt{EId}\ x : \Gamma(x)}$$

$$\frac{}{\Gamma \vdash \texttt{EInt}\ i : \texttt{int}} \qquad \frac{\Gamma(x) = \texttt{int}}{\Gamma \vdash \texttt{EPreIncr}\ x : \texttt{int}} \qquad \frac{\Gamma(x) = \texttt{int}}{\Gamma \vdash \texttt{EPostIncr}\ x : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{ELt}\ e_1\ e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash \texttt{EAnd}\ e_1\ e_2 : \texttt{bool}}$$

2. Write syntax-directed *interpretation* rules for the *expressions* of Question 1.

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

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

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

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

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

$$\frac{\gamma \vdash e_1 \Downarrow \langle \texttt{false}; \gamma' \rangle}{\gamma \vdash \texttt{EAnd}\ e_1\ e_2 \Downarrow \langle \texttt{false}; \gamma' \rangle} \qquad \frac{\gamma \vdash e_1 \Downarrow \langle \texttt{true}; \gamma' \rangle \qquad \gamma' \vdash e_2 \Downarrow \langle b; \gamma'' \rangle}{\gamma \vdash \texttt{EAnd}\ e_1\ e_2 \Downarrow \langle b; \gamma'' \rangle}$$

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!)

   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 = 0;
        ldc 0
        istore 0

        ;; int k = 0;
        ldc 0
        istore 1

        ;; while (k++ < 10))
L0:
        iload 1
        iinc 1 1
        ldc 10
        if_icmpge L1

        ;; int i = 0;
        ldc 0
        istore 2

        ;; while (i++ < k))
L2:
        iload 2
        iinc 2 1
        iload 1
        if_icmpge L0

        ;; (int) n ++;

        iinc 0 1
        goto L2
L1:
        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. (6p)

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)	
`if_icmpge` L	$(P, V, S.v.w)$	$\rightarrow$	(L, V, S)	if $v \geq w$
`if_icmpge` L	$(P, V, S.v.w)$	$\rightarrow$	$(P + 1, V, S)$	unless $v \geq w$
`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)	$\rightarrow$	$(P + 1, V, S.i)$	
`inc` a i	(P, V, S)	$\rightarrow$	$(P + 1, V[a := V(a) + i], S)$	

Question 6 (Functional languages):

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

$$\begin{array}{lll} x & & \text{identifier} \\ n & ::= 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots & \text{numeral} \\ e & ::= n \mid e + e \mid x \mid \lambda x \to e \mid e\, e & \text{expression} \\ t & ::= \mathsf{Int} \mid t \to t & \text{type} \end{array}$$

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) $k : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \vdash k\,(\lambda f \to f) + 1 : \mathsf{Int}$

(b) $x : \mathsf{Int} \to \mathsf{Int},\ g : \mathsf{Int} \vdash x\,(y + 1) : \mathsf{Int}$

(c) $f : (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int}) \vdash (\lambda i \to f\, i)\,(\lambda y \to f\,(\lambda h \to h)\, y) : \mathsf{Int} \to \mathsf{Int}$

(d) $h : \mathsf{Int} \to \mathsf{Int} \vdash \lambda y \to \lambda h \to (h + 1) + y : \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int})$

(e) $x : \mathsf{Int} \to \mathsf{Int} \vdash \lambda f \to f\,(1 + f\,(f\, x)) : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$

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

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

SOLUTION:

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