KURSUS: *Programmer som data*

KURSUSKODE: BSPRDAT1KU

KURSUSANSVARLIG: Niels Hallenberg (nh@itu.dk)

Programmer som data: Skriftlig eksamen

IT UNIVERSITET I KØBENHAVN

B.SC. SOFTWAREUDVIKLING, 5. SEMESTER

Name	**Email**
Adrian Valdemar Borup	adbo@itu.dk

6. januar 2024

Indhold

Jeg erklærer hermed at jeg selv har lavet hele denne eksamensbesvarelse uden hjælp fra andre.

0.0 Foregående kommentarer

Når jeg viser kodeændringer, vil det blive vist i samme format som en Git diff. Nedenfor ses et eksempel på formatet.

```
--- a/2022-jan/opgave-1/notes.md
+++ b/2022-jan/opgave-1/notes.md
@@ -1,2 +1,2 @@
 Gammel, ændret linje
-Fjernet eller modificeret linje før ændring
+Ny eller modificeret linje efter ændring
```

De første 2 linjer visen filen, der er blevet ændret. Derefter vises linjetallet på ændringen. Dernæst er selve ændringen.

Eksamenssæt 1

Januar 2022

1.1 Micro–ML: Sets

1.1.1 Lexer og parser

Nedenfor ses kodeændringen:

```
--- a/2022-jan/opgave-1/Fun/Absyn.fs
+++ b/2022-jan/opgave-1/Fun/Absyn.fs
@@ -11,3 +11,4 @@
    | Call of expr * expr
+   | Set of expr list (* Exam *)

--- a/2022-jan/opgave-1/Fun/FunLex.fsl
+++ b/2022-jan/opgave-1/Fun/FunLex.fsl
@@ -48,6 +48,10 @@
    | "<="             { LE }
+   | "++"             { UNION }
+   | '{'              { LBRACE }
+   | '}'              { RBRACE }
+   | ','              { COMMA }
    | '+'              { PLUS }

--- a/2022-jan/opgave-1/Fun/FunPar.fsy
+++ b/2022-jan/opgave-1/Fun/FunPar.fsy
@@ -12,21 +12,22 @@
 %token ELSE END FALSE IF IN LET NOT THEN TRUE
-%token PLUS MINUS TIMES DIV MOD
+%token PLUS MINUS TIMES DIV MOD UNION
 %token EQ NE GT LT GE LE
-%token LPAR RPAR
+%token LPAR RPAR LBRACE RBRACE COMMA
 %token EOF
 ...
 %left GT LT GE LE
-%left PLUS MINUS
```

```
+%left PLUS MINUS UNION
 %left TIMES DIV MOD
 ...
 %type <Absyn.expr> AppExpr
+%type <Absyn.expr list> SetItems

 %%

@@ -50,6 +51,7 @@ Expr:
    | Expr LE    Expr                    { Prim("<=", $1, $3)     }
+   | Expr UNION Expr                    { Prim("++", $1, $3)     }
 ;

 AtExpr:
@@ -57,6 +59,7 @@ AtExpr:
    | LET NAME NAME EQ Expr IN Expr END   { Letfun($2, $3, $5, $7) }
+   | LBRACE SetItems RBRACE              { Set($2)                }
    | LPAR Expr RPAR                      { $2                     }

@@ -69,3 +72,8 @@ Const:
    | CSTBOOL                             { CstB($1)               }
 ;
+
+SetItems:
+    SetItems COMMA Expr { $3 :: $1  }
+  | Expr                { [$1]      }
+;
```

Koden kompileres og køres med:

```
$ fslex --unicode FunLex.fsl
$ fsyacc --module FunPar FunPar.fsy
$ fsharpi -r FsLexYacc.Runtime.dll Absyn.fs FunPar.fs FunLex.fs Parse.fs Fun.fs
 ↪  ParseAndRun.fs
> open Parse;;
```

Herefter bliver resultatet for det første eksempel:

```
> fromString @"let s1 = {2, 3} in
-   let s2 = {1, 4} in
-     s1 ++ s2 = {2,4,3,1}
-   end
- end";;
val it : Absyn.expr =
  Let
    ("s1", Set [CstI 3; CstI 2],
     Let
       ("s2", Set [CstI 4; CstI 1],
```

```
        Prim
          ("=", Prim ("++", Var "s1", Var "s2"),
           Set [CstI 1; CstI 3; CstI 4; CstI 2])))
```

Og resultatet for den tomme mængde bliver, som specificeret, en fejl:

```
> fromString "let s = {} in s end";;
System.Exception: parse error near line 1, column 10

  at Microsoft.FSharp.Core.PrintfModule+PrintFormatToStringThenFail@1433[TResult
 ↪ ].Invoke (System.String message) [0x00000] in
 ↪ <b56f33d2f53c2e7533e6754e4d8591b5>:0
  ...
```

1.1.2 Evalueringstræ

Nedenfor ses evalueringstræet jf. kursusbogens Figur 4.3 samt de to nye evalueringsregler fra eksamenssættet.

$$\dfrac{\dfrac{}{\rho \vdash 1 \Rightarrow 1}\,(e1) \quad \dfrac{}{\rho \vdash 2 \Rightarrow 2}\,(e1)}{\rho \vdash \{1,\ 2\} \Rightarrow \mathsf{SetV}\{1,2\}}\,(set) \quad \dfrac{\dfrac{\rho'(\mathsf{s}) = \mathsf{SetV}\{1,2\}}{\rho' \vdash \mathsf{s} \Rightarrow \mathsf{SetV}\{1,2\}}\,(e3) \quad \dfrac{}{\rho' \vdash 3 \Rightarrow 3}\,(e1)}{\rho' \vdash 3 \Rightarrow \mathsf{SetV}\{3\}}\,(set)}{\rho' = \rho[\mathsf{s} \mapsto \mathsf{SetV}\{1,2\}] \vdash \mathsf{s}\ \texttt{++}\ \{3\} \Rightarrow \mathsf{SetV}\{1,2,3\}}\,(++)$$
$$\rho \vdash \texttt{let s = \{1, 2\} in s ++ \{3\} end} \Rightarrow \mathsf{SetV}\{1,2,3\}\,(e6)$$

1.1.3 Implementation

```
--- a/2022-jan/opgave-1/Fun/HigherFun.fs
+++ b/2022-jan/opgave-1/Fun/HigherFun.fs
@@ -30,11 +30,13 @@
 type value =
   | Int of int
   | Closure of string * string * expr * value env
+  | SetV of Set<value>

 let rec eval (e : expr) (env : value env) : value =
     match e with
     | CstI i -> Int i
     | CstB b -> Int (if b then 1 else 0)
+    | Set s -> SetV(s |> List.map (fun e' -> eval e' env) |> Set.ofList)
     | Var x  -> lookup env x

@@ -45,6 +47,8 @@
       | ("<", Int i1, Int i2) -> Int (if i1 < i2 then 1 else 0)
+      | ("++", SetV s1, SetV s2) -> SetV (Set.union s1 s2)
```

```
19 +       | ("=", SetV s1, SetV s2) -> Int (if s1 = s2 then 1 else 0)
20         |  _ -> failwith "unknown primitive or wrong type"
```

På linje 7 bliver `value`-typen udvidet med `SetV`-varianten, der opbevarer en F#-mængde internt.

På linje 13 findes koden til at evaluere en `Set`-expression, hvilket returnerer en instans af `SetV`. `Set` holder på en liste af expressions, som først skal evalueres, før den endelige `SetV` kan returneres — derfor bruges `List.map` til at evaluere hvert element i listen. Her er det vigtigt, at `List.map` sker før, listen bliver lavet om til en mængde. Hvis man bruger `map` på mængden i stedet for listen, kan man risikere at fjerne eksekveringen af visse udtryk, hvis der er flere af det samme udtryk. Hvis udtrykket har sideeffekter (såsom at skrive til skærmen), vil det kunne bemærkes.

På linje 18 og 19 implementeres operatorerne `++` og `=` for mængder ved hjælp af F#'s indbyggede konstruktioner. `Set.union` forener de to mængder, og `=` sammenligner rekursivt hvert element.

Forneden ses resultatet af at køre `ex01` gennem `HigherFuns` fortolker. Resultatet er 1, hvilket indikerer, at `s1 ++ s2` er lig mængden $\{1, 2, 3, 4\}$.

```
1  > fromString @"let s1 = {2, 3} in
2  -   let s2 = {1, 4} in
3  -     s1 ++ s2 = {2,4,3,1}
4  -   end
5  - end";;
6  val it : Absyn.expr =
7    Let
8      ("s1", Set [CstI 3; CstI 2],
9       Let
10        ("s2", Set [CstI 4; CstI 1],
11         Prim
12           ("=", Prim ("++", Var "s1", Var "s2"),
13            Set [CstI 1; CstI 3; CstI 4; CstI 2])))
14
15 > HigherFun.eval it [];;
16 val it : HigherFun.value = Int 1
```

1.1.4 Typeregler

Mine forslag til typereglerne er som følgende:

$$\frac{\rho \vdash e_1 : t \text{ set} \qquad \rho \vdash e_2 : t \text{ set}}{\rho \vdash e_1 \text{ ++ } e_2 : t \text{ set}} \ (\text{++}) \qquad \frac{\rho \vdash e_1 : t \text{ set} \qquad \rho \vdash e_2 : t \text{ set}}{\rho \vdash e_1 = e_2 : \text{bool}} \ (=)$$

Til ++-reglen har jeg antaget, at mængden kun kan indeholde elementer af samme type. Derfor har jeg i reglen specificeret, at hvis forener 2 mængder, så skal de begge have samme type t. Dermed bliver foreningsmængden også til typen t.

Til =-reglen har jeg samme antagelse som ovenfor. Det giver kun mening at sammenligne to mængder, hvis, hvis elementerne har samme type. Resultatet bliver sandt eller falsk alt efter om de to mængder er lig hinanden, hvilket repræsenteres med `bool`. I den faktiske implementation er typen `int`, idet returværdien er 0 eller 1 frem for sand eller falsk.

1.2 Micro–C: Print Stack

1.2.1 Lexer og parser

Jeg har ændret lexeren og parseren som følgende:

```
--- a/2022-jan/opgave-2/MicroC/Absyn.fs
+++ b/2022-jan/opgave-2/MicroC/Absyn.fs
@@ -35,6 +35,7 @@ and stmt =
   | Expr of expr                    (* Expression statement   e;   *)
   | Return of expr option           (* Return from method          *)
   | Block of stmtordec list         (* Block: grouping and scope   *)
+  | PrintStack of expr

 and stmtordec =
   | Dec of typ * string             (* Local variable declaration  *)

--- a/2022-jan/opgave-2/MicroC/CLex.fsl
+++ b/2022-jan/opgave-2/MicroC/CLex.fsl
@@ -25,6 +25,7 @@ let keyword s =
     | "print"   -> PRINT
     | "println" -> PRINTLN
     | "return"  -> RETURN
+    | "printStack"  -> PRINTSTACK
     | "true"    -> CSTBOOL 1
     | "void"    -> VOID
     | "while"   -> WHILE

--- a/2022-jan/opgave-2/MicroC/CPar.fsy
+++ b/2022-jan/opgave-2/MicroC/CPar.fsy
@@ -15,6 +15,7 @@ let nl = CstI 10
 %token <string> CSTSTRING NAME

 %token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE
+%token PRINTSTACK
 %token PLUS MINUS TIMES DIV MOD
 %token EQ NE GT LT GE LE
 %token NOT SEQOR SEQAND
@@ -95,6 +96,7 @@ Stmt:
```

```
34
35  StmtM:  /* No unbalanced if-else */
36      Expr SEMI                           { Expr($1)             }
37  +   | PRINTSTACK Expr SEMI               { PrintStack $2        }
38      | RETURN SEMI                       { Return None          }
39      | RETURN Expr SEMI                  { Return(Some($2))     }
40      | Block                             { $1                   }
```

Ved at parse `fac.c`, får man følgende abstrakte syntakstræ, hvor `printStack` ses på linje 25 og 33:

```
1  > open ParseAndRun;;
2  > fromFile "fac.c";;
3  val it : Absyn.program =
4    Prog
5      [Vardec (TypI, "nFac"); Vardec (TypI, "resFac");
6       Fundec
7         (None, "main", [(TypI, "n")],
8          Block
9            [Dec (TypI, "i"); Stmt (Expr (Assign (AccVar "i", CstI 0)));
10            Stmt (Expr (Assign (AccVar "nFac", CstI 0)));
11            Stmt
12              (While
13                 (Prim2 ("<", Access (AccVar "i"), Access (AccVar "n")),
14                  Block
15                    [Stmt
16                       (Expr
17                          (Assign
18                             (AccVar "resFac",
19                              Call ("fac", [Access (AccVar "i")])))));
20                   Stmt
21                     (Expr
22                        (Assign
23                           (AccVar "i",
24                            Prim2 ("+", Access (AccVar "i"), CstI 1))))]));
25            Stmt (PrintStack (CstI 42))]);
26       Fundec
27         (Some TypI, "fac", [(TypI, "n")],
28          Block
29            [Stmt
30               (Expr
31                  (Assign
32                    (AccVar "nFac", Prim2 ("+", Access (AccVar "nFac"), CstI 1))));
33             Stmt (PrintStack (Access (AccVar "nFac")));
34             Stmt
35               (If
36                  (Prim2 ("==", Access (AccVar "n"), CstI 0),
37                   Return (Some (CstI 1)),
```

```
                Return
                  (Some
                     (Prim2
                        ("*", Access (AccVar "n"),
                         Call
                           ("fac", [Prim2 ("-", Access (AccVar "n"), CstI
 ↪  1)])))))])])]
```

1.2.2 Machine.fs

```
--- a/2022-jan/opgave-2/MicroC/Machine.fs
+++ b/2022-jan/opgave-2/MicroC/Machine.fs
@@ -14,6 +14,7 @@ module Machine
 type label = string

 type instr =
+  | PRINTSTACK
   | Label of label             (* symbolic label; pseudo-instruc. *)
   | CSTI of int                (* constant                        *)
   | ADD                        (* addition                        *)
@@ -91,6 +92,7 @@ let CODEPRINTI = 22
 let CODEPRINTC = 23
 let CODELDARGS = 24
 let CODESTOP   = 25;
+let CODEPRINTSTACK = 26

 (* Bytecode emission, first pass: build environment that maps
    each label to an integer address in the bytecode.
@@ -98,6 +100,7 @@ let CODESTOP   = 25;

 let makelabenv (addr, labenv) instr =
     match instr with
+    | PRINTSTACK     -> (addr+1, labenv)
     | Label lab      -> (addr, (lab, addr) :: labenv)
     | CSTI i         -> (addr+2, labenv)
     | ADD            -> (addr+1, labenv)
@@ -130,6 +133,7 @@ let makelabenv (addr, labenv) instr =

 let rec emitints getlab instr ints =
     match instr with
+    | PRINTSTACK     -> CODEPRINTSTACK :: ints
     | Label lab      -> ints
     | CSTI i         -> CODECSTI   :: i :: ints
     | ADD            -> CODEADD    :: ints
```

1.2.3 `Machine.java`

Forneden ses kodeændringerne i `Machine.java` efterfulgt af en forklaring.

```
--- a/2022-jan/opgave-2/MicroC/Machine.java
+++ b/2022-jan/opgave-2/MicroC/Machine.java
@@ -35,7 +35,8 @@
     LDARGS = 24,
-    STOP = 25;
+    STOP = 25,
+    PRINTSTACK = 26;

@@ -127,6 +128,10 @@
       case PRINTC:
         System.out.print((char)(s[sp])); break;
+      case PRINTSTACK:
+        int v = s[sp--];
+        printStack(v, s, bp, sp);
+        break;
       case LDARGS:
@@ -140,6 +145,31 @@
   }

+  static void printStack(int v, int s[], int bp, int sp) {
+    System.out.format("-Print Stack %d----------------%n", v);
+
+    // Iterate over all stack frames
+    while (bp != -999) {
+      // Print current stack frame
+      System.out.println("Stack Frame");
+      for (int i = sp; i >= bp; i--) {
+        System.out.format(" s[%d]: Local/Temp = %d%n", i, s[i]);
+      }
+      System.out.format(" s[%d]: bp = %d%n", bp - 1, s[bp - 1]);
+      System.out.format(" s[%d]: ret = %d%n", bp - 2, s[bp - 2]);
+
+      // Move to next stack frame
+      sp = bp - 3;
+      bp = s[bp - 1];
+    }
+
+    // Print remaining items on the stack: global variables
+    System.out.println("Global");
+    for (int i = sp; i >= 0; i--) {
+      System.out.format(" s[%d]: %d%n", i, s[i]);
+    }
+  }
+
   // Print the stack machine instruction at p[pc]
```

```
46
47    static String insname(int[] p, int pc) {
48  @@ -170,6 +200,7 @@ class Machine {
49       case STOP:   return "STOP";
50  +    case PRINTSTACK: return "PRINTSTACK";
51       default:     return "<unknown>";
52       }
53    }
```

Linje 7 viser, at bytekoden 26 forbindes med operationen `PRINTSTACK`.

Linje 12-15 håndterer `PRINTSTACK`-instruktionen ved at poppe den øverste værdi af stakken (l. 13) og kalde en ny funktion `printStack` med denne værdi samt den nuværende tilstand af stakken.

Linje 20-43 viser koden for `printStack`-funktionen:

1. Først printes den poppede værdi (l. 21)
2. Der itereres over alle aktiveringsposter (l. 24). Det gøres ved at følge alle basepegere (l. 34-35), indtil basepegeren bliver `-999`, da vi på det tidspunkt er nået til `main`-funktionens aktiveringspost.
   (a) Print aktiveringsposten ved at gennemgå alle adresser mellem basepegeren og stakpegeren.
   (b) Print derefter `ret` og `bp` manuelt. Basepegeren peger på første lokale værdi i aktiveringsposten, og derfor må den gamle basepeger være på adressen lige før den nuværende basepeger. Og `ret` findes på adressen før den gamle basepeger.
   (c) Opdater basepegeren og stakpegeren, så vi kan læse den foregående aktiveringspost. Fordi de to adresser lige under den nuværende basepeger er allokeret til `ret` og den gamle basepeger, kan vi sætte den næste stakpeger til adressen 3 lokationer under basepegeren — altså toppen af den næste aktiveringspost.
3. Hvis der er defineret globale variable, findes de på adresserne mellem 0 og starten af `main`-funktionen. Derfor printer vi alle adresser efter den foregående iteration som globale variable.

1.2.4 `Comp.fs`

Ændringen ses her:

```
1  --- a/2022-jan/opgave-2/MicroC/Comp.fs
2  +++ b/2022-jan/opgave-2/MicroC/Comp.fs
3  @@ -144,6 +144,8 @@ let rec cStmt stmt (varEnv : varEnv) (funEnv : funEnv) :
   ↪  instr list =
4        [RET (snd varEnv - 1)]
5      | Return (Some e) ->
```

```
6         cExpr e varEnv funEnv @ [RET (snd varEnv)]
7   +   | PrintStack e ->
8   +     cExpr e varEnv funEnv @ [PRINTSTACK]
9
10  and cStmtOrDec stmtOrDec (varEnv : varEnv) (funEnv : funEnv) : varEnv * instr
    ↪  list =
11      match stmtOrDec with
```

Først kompileres udtrykket, der bruges som argument til `printStack`. Når dette er kompileret, vil udtrykkets endelige værdi findes på toppen af stakken ved køretid. Derfor mangler der blot selve `PRINTSTACK`-instruktionen, der popper værdien af stakken og bruger den, hvorfor den er tilføjet til sidst.

Bytekoden for `fac.c` ses forneden. Linje 10 og 12 viser instruktionerne til at printe stakken: linje 10 bruger den konstante værdi 42, og linje 12 indlæser værdien på adresse 0 (den globale variabel `nFac`).

```
1  > open ParseAndComp;;
2  > compile "fac";;
3  val it : Machine.instr list =
4    [INCSP 1; INCSP 1; LDARGS; CALL (1, "L1"); STOP; Label "L1"; INCSP 1; GETBP;
5     CSTI 1; ADD; CSTI 0; STI; INCSP -1; CSTI 0; CSTI 0; STI; INCSP -1;
6     GOTO "L4"; Label "L3"; CSTI 1; GETBP; CSTI 1; ADD; LDI; CALL (1, "L2"); STI;
7     INCSP -1; GETBP; CSTI 1; ADD; GETBP; CSTI 1; ADD; LDI; CSTI 1; ADD; STI;
8     INCSP -1; INCSP 0; Label "L4"; GETBP; CSTI 1; ADD; LDI; GETBP; CSTI 0; ADD;
9     LDI; LT; IFNZRO "L3";
10    CSTI 42; PRINTSTACK;
11    INCSP -1; RET 0; Label "L2"; CSTI 0; CSTI 0; LDI; CSTI 1; ADD; STI; INCSP -1;
12    CSTI 0; LDI; PRINTSTACK;
13    GETBP; CSTI 0; ADD; LDI; CSTI 0; EQ; IFZERO "L5"; CSTI 1; RET 1; GOTO "L6";
14    Label "L5"; GETBP; CSTI 0; ADD; LDI; GETBP; CSTI 0; ADD; LDI; CSTI 1; SUB;
15    CALL (1, "L2"); MUL; RET 1; Label "L6"; INCSP 0; RET 0]
```

Ved kørsel af `fac.c` gennem bytekodefortolkeren, fås følgende output:

```
1  $ java Machine.java fac.out 1
2  -Print Stack 1----------------
3  Stack Frame
4   s[9]: Local/Temp = 0
5   s[8]: bp = 4
6   s[7]: ret = 39
7  Stack Frame
8   s[6]: Local/Temp = 1
9   s[5]: Local/Temp = 0
10  s[4]: Local/Temp = 1
11  s[3]: bp = -999
12  s[2]: ret = 8
```

```
Global
 s[1]: 0
 s[0]: 1
-Print Stack 42----------------
Stack Frame
 s[5]: Local/Temp = 1
 s[4]: Local/Temp = 1
 s[3]: bp = -999
 s[2]: ret = 8
Global
 s[1]: 1
 s[0]: 1

Ran 0.005 seconds
```

1.2.5 Gennemgang af bytekode

```
INCSP 1;       // nFac som global variabel
INCSP 1;       // resFac som global variabel
LDARGS;        // Loade parameter n fra kommandolinie
CALL (1,"L1"); // Kalde main med n som argument.
STOP;          // Stop ved retur fra main.
Label "L1";                                    // Main
    INCSP 1;                                   // i som lokal variabel
    GETBP; CSTI 1; ADD; CSTI 0; STI; INCSP -1; // i = 0
    CSTI 0; CSTI 0; STI; INCSP -1;             // Sætte nFac = 0
    GOTO "L4";                                 // Hop til while-løkken
Label "L3";                       // While-løkkens krop
    CSTI 1;                       // Adresse på resFac
    GETBP; CSTI 1; ADD; LDI;      // Læg i på stakken som argument
    CALL (1,"L2"); STI; INCSP -1; // Kald fac(i) og gem resultatet i resFac (hvis
 ↪  adresse er på toppen af stakken)
    GETBP; CSTI 1; ADD;           // Beregn adressen på i
    GETBP; CSTI 1; ADD; LDI;      // Læs værdien af i
    CSTI 1; ADD; STI; INCSP -1;   // Gem i + 1 på i's adresse
    INCSP 0;                      // Ligegyldig oprydning pga. blokkens slutning
Label "L4";                   // While-løkken
    GETBP; CSTI 1; ADD; LDI; // Læs værdien af i
    GETBP; CSTI 0; ADD; LDI; // Læs værdien af n
    LT; IFNZRO "L3";         // Hvis i < n, gå ind i løkkens krop
    CSTI 42; PRINTSTACK;     // Når while-løkken er færdig, kaldes printStack med
 ↪  42 som argument
    INCSP -1;              // Oprydning på stakken (gammel basepeger deallokeres)
    RET 0;                   // Returnér
Label "L2";                      // Label til fac(int n)
    CSTI 0; CSTI 0; LDI; CSTI 1; // Adresse på nFac, værdi af nFac, og 1 lægges
 ↪  alle på stakken
```

```
    ADD; STI; INCSP -1;          // nFac = nFac + 1
    CSTI 0; LDI; PRINTSTACK;     // Kør printStack med nFac som argument
    GETBP; CSTI 0; ADD; LDI;     // Læs første argument: n
    CSTI 0; EQ; IFZERO "L5";     // Hvis n != 0, gå til L5
    CSTI 1; RET 1; GOTO "L6";    // Hvis n = 0, returnér 1
Label "L5";                              // False-case i fac
    GETBP; CSTI 0; ADD; LDI;             // Læs værdien af n
    GETBP; CSTI 0; ADD; LDI; CSTI 1; SUB; // Beregn n - 1
    CALL (1,"L2");                       // Kald fac med n - 1 som argument
    MUL; RET 1;                          // Gang resultat med n og returnér
Label "L6";        // Ubrugt kode til true-case i fac
    INCSP 0; RET 0 // Ubrugt
```

1.2.6 Forståelse af uddata

```
MicroC % java Machine fac.out 1
-Print Stack 1----------------
Stack Frame // Funktion fac
s[9]: Local/Temp = 0 // Stakplads til lokal variabel n
s[8]: bp = 4
s[7]: ret = 39
Stack Frame // Main
s[6]: Local/Temp = 1 // Temporær værdi: adressen på resFac, hvortil resultatet af
 ↪  fac(i) skal skrives
s[5]: Local/Temp = 0 // Lokal variabel i
s[4]: Local/Temp = 1 // Lokal variabel n
s[3]: bp = -999
s[2]: ret = 8
Global
s[1]: 0 // resFac
s[0]: 1 // nFac
-Print Stack 42----------------
Stack Frame // Main
s[5]: Local/Temp = 1 // Lokal variabel i
s[4]: Local/Temp = 1 // Lokal variabel n
s[3]: bp = -999
s[2]: ret = 8
Global
s[1]: 1 // resFac
s[0]: 1 // nFac
```

1.3 Micro–C: Intervalcheck

1.3.1 Lexer og parser

Jeg har ændret lexeren og parseren som vist nedenfor. `Absyn.fs` er udvidet med en type `Within`, der indeholder udtrykkene e, e_1 og e_2. Lexeren er udvidet, så strengen "within" oversættes til en `WITHIN`-token. Parseren er udvidet, så `WITHIN` har samme præcedens som andre logiske operatorer. Fordi den nye sprogfunktionalitet er et udtryk, er parser-reglen tiføjet som en del af `ExprNotAccess`.

```
--- a/2022-jan/opgave-3/MicroC/Absyn.fs
+++ b/2022-jan/opgave-3/MicroC/Absyn.fs
@@ -14,6 +14,7 @@ type typ =
   | TypP of typ                       (* Pointer type               *)

 and expr =
+  | Within of expr * expr * expr
   | Access of access                  (* x    or  *p    or  a[e]     *)
   | Assign of access * expr           (* x=e  or  *p=e  or  a[e]=e   *)
   | Addr of access                    (* &x   or  &*p   or  &a[e]    *)

--- a/2022-jan/opgave-3/MicroC/CLex.fsl
+++ b/2022-jan/opgave-3/MicroC/CLex.fsl
@@ -16,6 +16,7 @@ let lexemeAsString lexbuf =

 let keyword s =
     match s with
+    | "within"  -> WITHIN
     | "char"    -> CHAR
     | "else"    -> ELSE
     | "false"   -> CSTBOOL 0

--- a/2022-jan/opgave-3/MicroC/CPar.fsy
+++ b/2022-jan/opgave-3/MicroC/CPar.fsy
@@ -16,7 +16,7 @@ let nl = CstI 10

 %token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE
 %token PLUS MINUS TIMES DIV MOD
-%token EQ NE GT LT GE LE
+%token EQ NE GT LT GE LE WITHIN
 %token NOT SEQOR SEQAND
 %token LPAR RPAR LBRACE RBRACE LBRACK RBRACK SEMI COMMA ASSIGN AMP
 %token EOF
@@ -26,7 +26,7 @@ let nl = CstI 10
 %left SEQOR
 %left SEQAND
 %left EQ NE
-%left GT LT GE LE
+%left GT LT GE LE WITHIN
```

```
 %left PLUS MINUS
 %left TIMES DIV MOD
 %nonassoc NOT AMP
@@ -117,6 +117,7 @@ ExprNotAccess:
     AtExprNotAccess                              { $1                 }
   | Access ASSIGN Expr                           { Assign($1, $3)     }
   | NAME LPAR Exprs RPAR                         { Call($1, $3)       }
+  | Expr WITHIN LBRACK Expr COMMA Expr RBRACK { Within($1, $4, $6)  }
   | NOT Expr                                     { Prim1("!", $2)     }
   | PRINT Expr                                   { Prim1("printi", $2) }
   | PRINTLN                                      { Prim1("printc", nl) }
```

Ved at parse `within.c` med ovenstående ændringer fås følgende abstrakte syntakstræ:

```
> open ParseAndComp;;
> fromFile "within.c";;
val it : Absyn.program =
  Prog
    [Fundec
       (None, "main", [],
        Block
          [Stmt
             (Expr
                (Prim1
                   ("printi",
                    Within
                      (CstI 0, Prim1 ("printi", CstI 1),
                       Prim1 ("printi", CstI 2))))));
           Stmt
             (Expr
                (Prim1
                   ("printi",
                    Within
                      (CstI 3, Prim1 ("printi", CstI 1),
                       Prim1 ("printi", CstI 2))))));
           Stmt
             (Expr
                (Prim1
                   ("printi",
                    Prim1
                      ("printi",
                       Within
                         (CstI 42, Prim1 ("printi", CstI 40),
                          Prim1 ("printi", CstI 44)))))));
           Stmt
             (Expr
                (Prim1
                   ("printi",
```

```
35                    Within
36                      (Prim1 ("printi", CstI 42), Prim1 ("printi", CstI 40),
37                       Prim1 ("printi", CstI 44))))])]
```

1.3.2 Oversætterskema

Oversætterskemaet for `within` kan se således ud (med kommentarer til højre):

E[[e `within [` e_1 `,` e_2 `]`]] =

	E[[e]]	Oversæt e (med resultat v)
	E[[e_1]]	Oversæt e_1 (med resultat v_1)
	`SWAP`	Byt om på v_1 og v
	`DUP`	Duplikér v
	E[[e_2]]	Oversæt e_2 (med resultat v_2)
	`SWAP; LT; NOT`	Tjek $v \leq v_2$
	`IFZERO` cleanup	Hvis ikke $v \leq v_2$, fortsæt til oprydning
	`SWAP; LT; NOT`	Tjek $v_1 \leq v$
	`GOTO` end	Rutinen er færdig
cleanup:	`INCSP -2`	Deallokér v og v_1
	`CSTI 0`	Efterlad et 0 for at indikere false
end:	…	Rutinen er færdig

Det første, der sikres i ovenstående oversætterskema er, at de tre udtryk bliver ekseveret i rækkefølgen e, e_1, e_2, som det er specificeret. Ved brug af `SWAP` og `DUP` kommer stakken til at se således ud efter de første fem linjer: s, v_1, v, v, v_2.

Bytekodefortolkeren har ikke bytekode til direkte at lave et <=-tjek — der er kun en kode til <. Fordi `a <= b` er det samme sjom `!(b < a)`, kan vi beregne $\leq$ i bytekode med tre instruktioner i denne rækkefølge: `SWAP; LT; NOT`.

Derefter tjekkes det, om $v \leq v_2$. Hvis det er falsk, udføres en oprydning: efter $\leq$-tjekket, er v_1 og v stadig på stakken, og derfor bruges `INCSP -2` til at fjerne dem. Derefter efterlades 0 på toppen af stakken for at indikere, at $v_1 \leq v \leq v_2$ er falsk. Herefter er rutinen færdig.

Hvis $v \leq v_2$ derimod er sandt, tjekkes det om $v_1 \leq v$ også er sandt. Fordi disse to værdier er de eneste af vores værdier tilbage på stakken, vil stakken blot indeholde et 1 eller 0 afhængigt af $v_1 \leq v$. Derfor hopper vi til `end`, idet der ikke er mere at gøre.

Ovenstående opfylder de semantiske krav, fordi:

- De tre udtryk hver ekseveres præcis 1 gang
- Udtrykkende eksekveres i rækkefølgen e, e_1, e_2

- Der bliver efterladt enten et 1 eller 0 på stakken, og værdien er kun 1 hvis $v_1 \leq v \leq v_2$.

1.3.3 Implementering

Følgende diff viser ændringen til `Comp.fs`:

```
--- a/2022-jan/opgave-3/MicroC/Comp.fs
+++ b/2022-jan/opgave-3/MicroC/Comp.fs
@@ -206,6 +206,21 @@ and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) :
 ↪  instr list =
       @ cExpr e2 varEnv funEnv
       @ [GOTO labend; Label labtrue; CSTI 1; Label labend]
     | Call(f, es) -> callfun f es varEnv funEnv
+    | Within(e, e1, e2) ->
+        let labEnd = newLabel()
+        let labCleanup = newLabel()
+
+        cExpr e varEnv funEnv
+        @ cExpr e1 varEnv funEnv
+        @ [SWAP; DUP]
+        @ cExpr e2 varEnv funEnv
+        @ [SWAP; LT; NOT]
+        @ [IFZERO labCleanup]
+        @ [SWAP; LT; NOT]
+        @ [GOTO labEnd]
+        @ [Label labCleanup; INCSP -2; CSTI 0]
+        @ [Label labEnd]
```

Koden er struktureret på samme måde som det foregående oversætterskema, og forklaringen dertil gælder ligeledes for ovenstående kode.

Kompilerer man `within.c` og kører det igennem bytekodefortolkeren får man følgende uddata, som matcher det, der er givet af opgaven:

```
$ java Machine.java within.out
1 2 0 1 2 0 40 44 1 1 42 40 44 1
Ran 0.001 seconds
```

For komplethedens skyld er den genererede bytekode indsat her:

```
> open ParseAndComp;;
> compile "within";;
val it : Machine.instr list =
  [LDARGS; CALL (0, "L1"); STOP; Label "L1"; CSTI 0; CSTI 1; PRINTI; SWAP; DUP;
   CSTI 2; PRINTI; SWAP; LT; NOT; IFZERO "L3"; SWAP; LT; NOT; GOTO "L2";
   Label "L3"; INCSP -2; CSTI 0; Label "L2"; PRINTI; INCSP -1; CSTI 3; CSTI 1;
   PRINTI; SWAP; DUP; CSTI 2; PRINTI; SWAP; LT; NOT; IFZERO "L5"; SWAP; LT;
   NOT; GOTO "L4"; Label "L5"; INCSP -2; CSTI 0; Label "L4"; PRINTI; INCSP -1;
```

```
    CSTI 42; CSTI 40; PRINTI; SWAP; DUP; CSTI 44; PRINTI; SWAP; LT; NOT;
    IFZERO "L7"; SWAP; LT; NOT; GOTO "L6"; Label "L7"; INCSP -2; CSTI 0;
    Label "L6"; PRINTI; PRINTI; INCSP -1; CSTI 42; PRINTI; CSTI 40; PRINTI;
    SWAP; DUP; CSTI 44; PRINTI; SWAP; LT; NOT; IFZERO "L9"; SWAP; LT; NOT;
    GOTO "L8"; Label "L9"; INCSP -2; CSTI 0; Label "L8"; PRINTI; INCSP -1;
    INCSP 0; RET -1]
```

1.4 Icon

1.4.1 1-10

```
> run (Every(Write (FromTo (1, 10))));;
1 2 3 4 5 6 7 8 9 10 val it : value = Int 0
```

Konstruktionen `FromTo(1, 10)` bruges til at generere en sekvens fra 1 til 10. Herefter bruges `Write` til at printe et element fra sekvensen. Men for at printe *alle* elementer skal konstruktionen `Every` bruges.

1.4.2 10-tals-tabellen

```
> run (Every(Write (Prim("*", FromTo(1, 10), FromTo(1, 10)))));;
1 2 3 4 5 6 7 8 9 10 2 4 6 8 10 12 14 16 18 20 3 6 9 12 15 18 21 24 27 30 4 8 12
 ↪  16 20 24 28 32 36 40 5 10 15 20 25 30 35 40 45 50 6 12 18 24 30 36 42 48 54 60
 ↪  7 14 21 28 35 42 49 56 63 70 8 16 24 32 40 48 56 64 72 80 9 18 27 36 45 54 63
 ↪  72 81 90 10 20 30 40 50 60 70 80 90 100 val it : value = Int 0
```

Igen gør jeg brug af `FromTo` til at generere en sekvens fra 1 til 10. Jeg bruger `Prim("*", a, b)` til at gange de to sekvenser sammen. For hvert tal i a_i i a bliver alle tal i sekvens b ganget med a_i og tilføjet som resultat i en ny sekvens. Derfor bliver resultatet `length(a) * length(b)` langt. Fordi jeg bruger to sekvenser fra 1-10, kommer alle tal fra 1-10 til at blive ganget med 1, så alle tal ganget med 2 osv.

1.4.3 10-tals-tabellen på flere linjer

```
> run (Every(Write (Prim("*", FromTo(1, 10), And(Write (CstS "\n"), FromTo(1,
 ↪  10))))));;

 1 2 3 4 5 6 7 8 9 10
 2 4 6 8 10 12 14 16 18 20
 3 6 9 12 15 18 21 24 27 30
 4 8 12 16 20 24 28 32 36 40
 5 10 15 20 25 30 35 40 45 50
```

```
 6 12 18 24 30 36 42 48 54 60
 7 14 21 28 35 42 49 56 63 70
 8 16 24 32 40 48 56 64 72 80
 9 18 27 36 45 54 63 72 81 90
 10 20 30 40 50 60 70 80 90 100 val it : value = Int 0
```

Koden er næsten det samme som for 10-tals-tabellen på én linje, men her bliver `And` brugt. `And(a, b)` producerer hvert element i `b` for hvert element i `a`. Jeg bruger det til at `Write` en ny linje (`CstS "\n"`) for hver gennemgang af de 10 tal på højre side.

1.4.4 Tilfældige tal

Ændringerne i `Icon.fs` kan ses forneden. Linje 7 viser, de nye variant, der holder de 3 givne parametre. Linje 26-34 viser hovedændringen. Her har jeg lavet en rekursiv funktion, der gør brug af den givne continuation. Rekursionen starter med argumentet 1 og plusser en til argumentet for hvert element, hvorved den stopper, når argumentet overskrider `num`. Når dette sker, bliver fejl-continuationen kaldt. Det er kaldet til continuationen, der bliver returneret (hvor det rekursive kald er lazy-evalueret).

```
--- a/2022-jan/opgave-4/Cont/Icon.fs
+++ b/2022-jan/opgave-4/Cont/Icon.fs
@@ -35,6 +35,7 @@ type expr =
    | Or  of expr * expr
    | Seq of expr * expr
    | Every of expr
+   | Random of int * int * int
    | Fail;;

 (* Runtime values and runtime continuations *)
@@ -54,6 +55,10 @@ let write v =
     | Int i -> printf "%d " i
     | Str s -> printf "%s " s;;

+let random = new System.Random();
+let randomNext(min, max) =
+    random.Next(min,max+1) // max is exclusive in Next.
+
 (* Expression evaluation with backtracking *)

 let rec eval (e : expr) (cont : cont) (econt : econt) =
@@ -90,6 +95,15 @@ let rec eval (e : expr) (cont : cont) (econt : econt) =
           econt
     | And(e1, e2) ->
       eval e1 (fun _ -> fun econt1 -> eval e2 cont econt1) econt
+    | Random(min, max, num) ->
+      let rec generateRandoms i =
+          if i <= num then
+              let rand = randomNext(min, max)
```

```
+              cont (Int rand) (fun () -> generateRandoms (i+1))
+          else
+              econt ()
+
+      generateRandoms 1
      | Or(e1, e2) ->
        eval e1 cont (fun () -> eval e2 cont econt)
      | Seq(e1, e2) ->
```

Forneden ses det givne eksempel kørt 5 gange, hvor det bekræftes, at tilfældige tal genereres.

```
> open Icon;;
> run (Every(Write(Random(1,10,3))));;
5 7 2 val it : value = Int 0

> run (Every(Write(Random(1,10,3))));;
6 8 7 val it : value = Int 0

> run (Every(Write(Random(1,10,3))));;
10 1 8 val it : value = Int 0

> run (Every(Write(Random(1,10,3))));;
9 8 4 val it : value = Int 0

> run (Every(Write(Random(1,10,3))));;
3 8 6 val it : value = Int 0
```

Eksamenssæt 2

Januar 2021

2.3 List–C: Tabeller på hoben

2.3.1 Abstrakt syntaks

```
--- a/2021-jan/opgave3/ListC/Absyn.fs
+++ b/2021-jan/opgave3/ListC/Absyn.fs
@@ -24,2 +24,3 @@ and expr =
   | Prim2 of string * expr * expr    (* Binary primitive operator   *)
+  | Prim3 of string * expr * expr * expr
   | Andalso of expr * expr           (* Sequential and              *)
```

```
> Stmt (Expr (Prim3 ("updateTable", Access (AccVar "t"), CstI 0, CstI 42)));;
val it : stmtordec =
  Stmt (Expr (Prim3 ("updateTable", Access (AccVar "t"), CstI 0, CstI 42)))
```

2.3.2 Lexer og parser

```
--- a/2021-jan/opgave3/ListC/CLex.fsl
+++ b/2021-jan/opgave3/ListC/CLex.fsl
@@ -34,2 +34,6 @@ let keyword s =
     | "setcdr"  -> SETCDR
+    | "createTable" -> CREATETABLE
+    | "updateTable" -> UPDATETABLE
+    | "indexTable" -> INDEXTABLE
+    | "printTable" -> PRINTTABLE
     | "true"    -> CSTBOOL 1

--- a/2021-jan/opgave3/ListC/CPar.fsy
+++ b/2021-jan/opgave3/ListC/CPar.fsy
@@ -17,2 +17,3 @@ let nl = CstI 10
```

```
 %token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE
+%token CREATETABLE UPDATETABLE INDEXTABLE PRINTTABLE
 %token NIL CONS CAR CDR DYNAMIC SETCAR SETCDR
@@ -148,2 +149,6 @@ AtExprNotAccess:
    | SETCDR LPAR Expr COMMA Expr RPAR        { Prim2("setcdr",$3,$5) }
+   | CREATETABLE LPAR Expr RPAR              { Prim1("createTable", $3)         }
+   | UPDATETABLE LPAR Expr COMMA Expr COMMA Expr RPAR { Prim3("updateTable", $3,
 ↪  $5, $7) }
+   | INDEXTABLE LPAR Expr COMMA Expr RPAR    { Prim2("indexTable", $3, $5)      }
+   | PRINTTABLE LPAR Expr RPAR               { Prim1("printTable", $3)          }
  ;
```

```
$ mono listcc.exe exam10.lc
List-C compiler v 1.0.0.0 of 2012-02-13
Compiling exam10.lc to exam10.out
Parsed: Prog
  [Fundec
     (None, "main", [],
      Block
        [Dec (TypD, "t");
         Stmt (Expr (Assign (AccVar "t", Prim1 ("createTable", CstI 3))));
         Stmt
           (Expr (Prim3 ("updateTable", Access (AccVar "t"), CstI 0, CstI 42)));
         Stmt
           (Expr (Prim3 ("updateTable", Access (AccVar "t"), CstI 1, CstI 43)));
         Stmt
           (Expr (Prim3 ("updateTable", Access (AccVar "t"), CstI 2, CstI 44)));
         Stmt
           (Expr
              (Prim1
                 ("printi", Prim2 ("indexTable", Access (AccVar "t"), CstI 0))));
         Stmt
           (Expr
              (Prim1
                 ("printi", Prim2 ("indexTable", Access (AccVar "t"), CstI 1))));
         Stmt
           (Expr
              (Prim1
                 ("printi", Prim2 ("indexTable", Access (AccVar "t"), CstI 2))));
         Stmt (Expr (Prim1 ("printTable", Access (AccVar "t"))))])]
ERROR: unknown primitive 1
```

2.3.3 Implementering af bytekodefortolker

```
--- a/2021-jan/opgave3/ListC/ListVM/ListVM/listmachine.c
+++ b/2021-jan/opgave3/ListC/ListVM/ListVM/listmachine.c
@@ -81,7 +81,7 @@ created when allocating all but the last word of a free block.
   #define PPCOMP "Not compiled with gcc."
 #endif

-#if _WIN64 || __x86_64__ || __ppc64__
+#if _WIN64 || __x86_64__ || __ppc64__ || __aarch64__
   #define ENV64
   #define PPARCH "64 bit architecture."
 #else
@@ -151,6 +151,7 @@ word* readfile(char* filename);
 #endif

 #define CONSTAG 0
+#define TABLETAG 1

 // Heap size in words

@@ -195,6 +196,10 @@ word *freelist;
 #define CDR 29
 #define SETCAR 30
 #define SETCDR 31
+#define CREATETABLE 32
+#define UPDATETABLE 33
+#define INDEXTABLE 34
+#define PRINTTABLE 35

 #define STACKSIZE 1000

@@ -237,6 +242,10 @@ void printInstruction(word p[], word pc) {
   case CDR:    printf("CDR"); break;
   case SETCAR: printf("SETCAR"); break;
   case SETCDR: printf("SETCDR"); break;
+  case CREATETABLE: printf("CREATETABLE"); break;
+  case UPDATETABLE: printf("UPDATETABLE"); break;
+  case INDEXTABLE: printf("INDEXTABLE"); break;
+  case PRINTTABLE: printf("PRINTTABLE"); break;
   default:     printf("<unknown>"); break;
   }
 }
@@ -379,6 +388,63 @@ int execcode(word p[], word s[], word iargs[], int iargc,
 ↪  int /* boolean */ trac
       word* p = (word*)s[sp];
       p[2] = v;
     } break;
```

```
+
+    case CREATETABLE: {
+      word N = Untag((word) s[sp--]);
+
+      if (N < 0) {
+        printf("Cannot create table with negative size\n");
+        return -1;
+      }
+
+      word* t = allocate(TABLETAG, N + 1, s, sp);
+
+      t[1] = N;
+      for (int i = 2; i <= N + 1; i++) {
+        t[i] = 0;
+      }
+
+      s[++sp] = (word) t;
+    } break;
+
+    case UPDATETABLE: {
+      word v = Untag((word) s[sp--]);
+      word i = Untag((word) s[sp--]);
+      word* t = (word*) s[sp];
+      word N = t[1];
+
+      if (i < 0 || i >= N) {
+        printf("Index %lld out of bounds\n", i);
+        return -1;
+      }
+
+      t[i + 2] = v;
+    } break;
+
+    case INDEXTABLE: {
+      word i = Untag((word) s[sp--]);
+      word* t = (word*) s[sp--];
+      word N = t[1];
+
+      if (i < 0 || i >= N) {
+        printf("Index %lld out of bounds\n", i);
+        return -1;
+      }
+
+      s[++sp] = Tag(t[i + 2]);
+    } break;
+    case PRINTTABLE: {
+      word* t = (word*) s[sp];
+      word N = t[1];
+
```

```
+      printf("[");
+      for (int i = 2; i < N + 1; i++)
+        printf("%lld, ", t[i]);
+      if (N > 0)
+        printf("%lld", t[N + 1]);
+      printf("]\n");
+    } break;
+
     default:
       printf("Illegal instruction " WORD_FMT " at address " WORD_FMT "\n",
              p[pc - 1], pc - 1);
```

2.3.4 Udvidelse af oversætteren

```
--- a/2021-jan/opgave3/ListC/Comp.fs
+++ b/2021-jan/opgave3/ListC/Comp.fs
@@ -180,7 +180,9 @@ and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) :
 ↪  instr list =
          | "printc" -> [PRINTC]
          | "car"    -> [CAR]
          | "cdr"    -> [CDR]
+         | "createTable" -> [CREATETABLE]
+         | "printTable" -> [PRINTTABLE]
          | _        -> raise (Failure "unknown primitive 1"))
     | Prim2(ope, e1, e2) ->
       cExpr e1 varEnv funEnv
       @ cExpr e2 varEnv funEnv
@@ -199,7 +201,16 @@ and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) :
 ↪  instr list =
          | "cons"   -> [CONS]
          | "setcar" -> [SETCAR]
          | "setcdr" -> [SETCDR]
+         | "indexTable" -> [INDEXTABLE]
          | _        -> raise (Failure "unknown primitive 2"))
+    | Prim3(ope, e1, e2, e3) ->
+      cExpr e1 varEnv funEnv
+      @ cExpr e2 varEnv funEnv
+      @ cExpr e3 varEnv funEnv
+      @ (match ope with
+         | "updateTable" -> [UPDATETABLE]
+         | _        -> raise (Failure "unknown primitive 3"))
     | Andalso(e1, e2) ->
       let labend   = newLabel()
       let labfalse = newLabel()

--- a/2021-jan/opgave3/ListC/Machine.fs
```

```
+++ b/2021-jan/opgave3/ListC/Machine.fs
@@ -47,6 +47,12 @@ type instr =
    | CDR                               (* get second field of cons cell   *)
    | SETCAR                            (* set first field of cons cell    *)
    | SETCDR                            (* set second field of cons cell   *)
+   | CREATETABLE
+   | UPDATETABLE
+   | INDEXTABLE
+   | PRINTTABLE

 (* Generate new distinct labels *)

@@ -103,6 +109,10 @@ let CODECAR    = 28;
 let CODECDR    = 29;
 let CODESETCAR = 30;
 let CODESETCDR = 31;
+let CODECREATETABLE = 32
+let CODEUPDATETABLE = 33
+let CODEINDEXTABLE = 34
+let CODEPRINTTABLE = 35

 (* Bytecode emission, first pass: build environment that maps
    each label to an integer address in the bytecode.
@@ -143,6 +153,10 @@ let makelabenv (addr, labenv) instr =
     | CDR            -> (addr+1, labenv)
     | SETCAR         -> (addr+1, labenv)
     | SETCDR         -> (addr+1, labenv)
+    | CREATETABLE    -> (addr+1, labenv)
+    | UPDATETABLE    -> (addr+1, labenv)
+    | INDEXTABLE     -> (addr+1, labenv)
+    | PRINTTABLE     -> (addr+1, labenv)

 (* Bytecode emission, second pass: output bytecode as integers *)

@@ -181,6 +195,10 @@ let rec emitints getlab instr ints =
     | CDR            -> CODECDR    :: ints
     | SETCAR         -> CODESETCAR :: ints
     | SETCDR         -> CODESETCDR :: ints
+    | CREATETABLE    -> CODECREATETABLE :: ints
+    | UPDATETABLE    -> CODEUPDATETABLE :: ints
+    | INDEXTABLE     -> CODEINDEXTABLE :: ints
+    | PRINTTABLE     -> CODEPRINTTABLE :: ints


 (* Convert instruction list to int list in two passes:
```

2.3.5 Uddata

Ved kørsel af programmet fås:

```
$ mono listcc.exe exam10.lc
List-C compiler v 1.0.0.0 of 2012-02-13
Compiling exam10.lc to exam10.out

$ gcc -Wall -g -o listmachine ./ListVM/ListVM/listmachine.c

$ ./listmachine exam10.out
42 43 44 [42, 43, 44]

Used 0 cpu milli-seconds
```

2.4 MicroML: Doubles

2.4.1 Abstrakt syntaks

Ændringen er som vist:

```
--- a/2021-jan/opgave4/Fun/Absyn.fs
+++ b/2021-jan/opgave4/Fun/Absyn.fs
@@ -4,6 +4,7 @@ module Absyn

 type expr =
    | CstI of int
+   | CstD of double
    | CstB of bool
    | Var of string
    | Let of string * expr * expr
```

Den abstrakte syntaks `exam01` accepteres, som det vises her:

```
> open Absyn;;
> Letfun("f1", "x", Prim("+", Var "x", CstD 1.0), Call(Var "f1", CstD 12.0));;
val it : expr =
  Letfun
    ("f1", "x", Prim ("+", Var "x", CstD 1.0), Call (Var "f1", CstD 12.0))
```

2.4.2 Implementering

Ændringen er som vist:

```
--- a/2021-jan/opgave4/Fun/HigherFun.fs
+++ b/2021-jan/opgave4/Fun/HigherFun.fs
@@ -29,11 +29,13 @@ let rec lookup env x =

 type value =
   | Int of int
+  | Double of double
   | Closure of string * string * expr * value env (* (f, x, fBody, fDeclEnv) *)

 let rec eval (e : expr) (env : value env) : value =
     match e with
     | CstI i -> Int i
+    | CstD d -> Double d
     | CstB b -> Int (if b then 1 else 0)
     | Var x  -> lookup env x
     | Prim(ope, e1, e2) ->
@@ -45,6 +47,11 @@ let rec eval (e : expr) (env : value env) : value =
       | ("-", Int i1, Int i2) -> Int (i1 - i2)
       | ("=", Int i1, Int i2) -> Int (if i1 = i2 then 1 else 0)
       | ("<", Int i1, Int i2) -> Int (if i1 < i2 then 1 else 0)
+      | ("*", Double d1, Double d2) -> Double (d1 * d2)
+      | ("+", Double d1, Double d2) -> Double (d1 + d2)
+      | ("-", Double d1, Double d2) -> Double (d1 - d2)
+      | ("=", Double d1, Double d2) -> Int (if d1 = d2 then 1 else 0)
+      | ("<", Double d1, Double d2) -> Int (if d1 < d2 then 1 else 0)
       |  _ -> failwith "unknown primitive or wrong type"
     | Let(x, eRhs, letBody) ->
       let xVal = eval eRhs env
```

`value`-typen er udvidet, så vi kan processere kommatal. Abstrakt syntaks `CstD` oversættes direkte til en `Double`. Operationerne er det samme for kommatal som for heltal, blot med forskellige typer. Dog returneres et heltal stadig for = og =, fordi boolske værdier (1 og 0) er repræsenteret som heltal.

Forneden vises kørsel af både `exam01` og `exam02Err`:

```
> open Absyn;;
> open HigherFun;;
> let exam01 = Letfun("f1", "x", Prim("+", Var "x", CstD 1.0), Call(Var "f1",
 ↪  CstD 12.0));- ;
val exam01 : expr =
  Letfun
    ("f1", "x", Prim ("+", Var "x", CstD 1.0), Call (Var "f1", CstD 12.0))

> eval exam01 [];;
val it : value = Double 13.0

> let exam02Err = Prim("+", CstD 23.0, CstI 1);;
```

```
val exam02Err : expr = Prim ("+", CstD 23.0, CstI 1)

> eval exam02Err [];;
System.Exception: unknown primitive or wrong type
  at FSI_0002.HigherFun.eval (FSI_0002.Absyn+expr e,
 ↪  Microsoft.FSharp.Collections.FSharpList`1[T] env) [0x00152] in
 ↪  <853e40332b7440d7bc75e1fda463f455>:0
  at <StartupCode$FSI_0011>.$FSI_0011.main@ () [0x0000a] in
 ↪  <853e40332b7440d7bc75e1fda463f455>:0
  at (wrapper managed-to-native) System.Reflection.RuntimeMethodInfo.InternalInv
 ↪  oke(System.Reflection.RuntimeMethodInfo,object,object[],System.Exception&)
  at System.Reflection.RuntimeMethodInfo.Invoke (System.Object obj,
 ↪  System.Reflection.BindingFlags invokeAttr, System.Reflection.Binder binder,
 ↪  System.Object[] parameters, System.Globalization.CultureInfo culture)
 ↪  [0x0006a] in <b27839cc2dba4804baacf2f5cce6de32>:0
Stopped due to error
```

2.4.3 Type casts

```
--- a/2021-jan/opgave4/Fun/Absyn.fs
+++ b/2021-jan/opgave4/Fun/Absyn.fs
@@ -11,2 +11,3 @@ type expr =
   | Prim of string * expr * expr
+  | Prim1 of string * expr
   | If of expr * expr * expr

--- a/2021-jan/opgave4/Fun/HigherFun.fs
+++ b/2021-jan/opgave4/Fun/HigherFun.fs
@@ -55,2 +55,8 @@ let rec eval (e : expr) (env : value env) : value =
       |  _ -> failwith "unknown primitive or wrong type"
+    | Prim1(ope, e) ->
+      let v = eval e env
+      match (ope, v) with
+      | ("toInt", Double d) -> Int (int d)
+      | ("toDouble", Int i) -> Double (double i)
+      |  _ -> failwith "unknown unary primitive or wrong type"
     | Let(x, eRhs, letBody) ->
```

`Prim1`-varianten er stort set den samme som `Prim`-varianten, men der evalueres kun 1 argument. Løsningen bruger F#'s type-casts til at implementere de to nye primitiver og giver en fejl, hvis typerne er forkerte eller en ukendt primitiv er givet.

Her ses det, at `exam05` korrekt evalueres til 46 samt at mit eget testeksempel fungerer:

```
> open Absyn;;
> open HigherFun;;
> let exam05 = Let("x",CstD 23.0, Prim("*",CstI 2,Prim1("toInt", Var "x")));;
val exam05 : expr =
  Let ("x", CstD 23.0, Prim ("*", CstI 2, Prim1 ("toInt", Var "x")))

> eval exam05 [];;
val it : value = Int 46

> let testEx = Let ("x", CstI -100, Prim ("-", Prim1 ("toDouble", Var "x"),  CstD
 ↪  2.0));;
val testEx : expr =
  Let ("x", CstI -100, Prim ("-", Prim1 ("toDouble", Var "x"), CstD 2.0))

> eval testEx [];;
val it : value = Double -102.0
```

2.4.4 Typeinferenstræ

$$\dfrac{\dfrac{}{\rho \vdash \texttt{34:int}}\,(p1)}{\rho \vdash \texttt{toDouble(34):double}}\,(toDouble) \qquad \dfrac{\dfrac{}{\rho \vdash \texttt{2:int}}\,(p1) \qquad \dfrac{\dfrac{}{\rho \vdash \texttt{5.0:double}}\,(cstD)}{\rho \vdash \texttt{toInt(5.0):int}}\,(toInt)}{\rho \vdash \texttt{2 * toInt(5.0):int}}\,(multI)}{\rho \vdash \texttt{toDouble(2 * toInt(5.0)):double}}\,(toDouble)$$

$$\rho \vdash \texttt{toDouble(34) * toDouble(2 * toInt(5.0)):double}\;(multD)$$

Eksamenssæt 3

December 2019

3.1 Regulære udtryk og automater

3.1.1 Årsager til at automaten er ikke–deterministisk

Automaten er ikke deterministisk fordi:

1. Der går 2 a-kanter ud fra knude 1
2. Der går 2 b-kanter ud fra knude 2
3. Der går en ϵ-kant fra knude 3 til knude 4

3.1.2 Eksempler på genkendelige strenge

Disse strenge bliver genkendt af automaten:

1. ab
2. abb
3. aaaaab

Der findes uendeligt mange flere.

3.1.3 Beskrivelse af sproget

Sproget, som automaten genkender, starter altid med et a. Derefter kommer der muligvis et b. Derefter kommer der mellem 0 til uendeligt a'er. Til sidst kommer der altid et b.

3.1.4 Konvertering til DFA

Forneden ses en systematisk konstruktion af DFA'en jf. Introduction to Compiler Design samt David Christiansens vejledning. Et × indikerer, at der ikke findes en overgang. En tilstand med streg under indikerer en accepterende tilstand.

DFA-tilstand	Ved a	Ved b	NFA-tilstande
s_1	s_2	×	$\{1\}$
s_2	s_3	s_4	$\{2,3,4\}$
s_3	s_3	s_5	$\{4\}$
$\underline{s_4}$	s_3	s_5	$\{3,4,\underline{5}\}$
$\underline{s_5}$	×	×	$\{\underline{5}\}$

Den konstruerede DFA er tegnet således: