Opgave 1

Udvid lexer og parser

Der er lavet følgende ændringer i `FunLex.fsl`

```
rule Token = parse
  ...
  | "++"             { UNION }
  | '{'              { LBRA }
  | '}'              { RBRA }
  | ','              { COMMA }
  ...
  | _                { failwith "Lexer error: illegal symbol" }
```

Der er lavet følgende ændringer i `FunPar.fsy`

```
%token COMMA LBRA RBRA UNION

%left ELSE              /* lowest precedence  */
%left EQ NE
%left GT LT GE LE
%left PLUS MINUS UNION
%left TIMES DIV MOD
%nonassoc NOT           /* highest precedence  */

Expr:
  ...
  | Expr PLUS  Expr                    { Prim("+",  $1, $3)     }
  | Expr UNION  Expr                   { Prim("++",  $1, $3)    }
  ...
  | LBRA Lst RBRA                      { Set($2)                 }
;

Lst:
    Expr                              { [$1] }
  | Expr COMMA Lst                    { $1 :: $3}
;
```

Angiv evalueringstræ

Nej

Udvid higherfun

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

```
  | SetV of Set<value> (* Exam *)

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)
    | Var x  -> lookup env x
    | Prim(ope, e1, e2) ->
      let v1 = eval e1 env
      let v2 = eval e2 env
      match (ope, v1, v2) with
      ...
      | ("++",SetV s1, SetV s2) ->
        (* Anvender den eksisterende union operation *)
        SetV(Set.union s1 s2)
      ...
      | ("=", SetV s1, SetV s2) ->
        (* Hvis de begge er et subset af hinanden så må de være det præcis samme sæt *)
        Int (if Set.isSubset s1 s2 && Set.isSubset s2 s1 then 1 else 0)
      ...
      |  _ -> failwith "unknown primitive or wrong type"
    ...
    | Set(lst) ->
      (*
        Da vi skal have et sæt af værdier skal vi evaluerer listen af expressions
        Dette kan vi herefter konvertere til et sæt og smide ind i SetV værdien
      *)
      lst
      |> List.map (fun e -> eval e env)
      |> Set.ofList
      |> SetV
```

Kørsel af programmet

```
opgave1 git:(main) dotnet run
"
        let s1 = {2, 3} in
            let s2 = s1++{1, 4} in
                s1 ++ s2 = {2, 4, 3, 1}
            end
        end
    "


Let
  ("s1", Set [CstI 2; CstI 3],
   Let
```

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

```
Int 1
```

Udvid typereglerne

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

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

Det giver ikke mening at sammenligne eller forene to sæt med forskellige typer, på samme måde som det ikke giver mening at lægge en bool og en int sammen eller at tjekke om en bool er lig en int. Derfor skal begge to være af samme type når vi udfører operationer på dem. Det skal dog siges at der ikke er noget typemæssigt der afholder nogen fra at placere flere forskellige typer i den nuværende implementation.

Derfor skulle et typsystem påkræve at begge sæt er ens fra et type perspektiv. Forenelses operation vil returnere et sæt som et de to sæt forenede, da vi lige har defineret at de skal være af samme type, så vil det nye sæt også være af den type.

Når der laves en sammenligning vil der altid skulle returneres en boolsk værdi, derfor returnere (=) typen *bool*.

Opgave 2

Udvid CLex CPar Absyn

CLex

```
let keyword s =
    match s with
    ...
    | "printStack" -> PRINTSTACK
    | _          -> NAME s
```

CPar

```
%token PRINTSTACK
```

```
%right ASSIGN             /* lowest precedence */
%nonassoc PRINT PRINTSTACK
%left SEQOR
%left SEQAND
%left EQ NE
%left GT LT GE LE
%left PLUS MINUS
%left TIMES DIV MOD
%nonassoc NOT AMP
%nonassoc LBRACK          /* highest precedence  */

StmtM:  /* No unbalanced if-else */
    Expr SEMI                           { Expr($1)             }
  | RETURN SEMI                         { Return None          }
  | RETURN Expr SEMI                    { Return(Some($2))     }
  | Block                               { $1                   }
  | IF LPAR Expr RPAR StmtM ELSE StmtM  { If($3, $5, $7)       }
  | WHILE LPAR Expr RPAR StmtM          { While($3, $5)        }
  | PRINTSTACK Expr SEMI                { PrintStack($2)      }
;
```

Absyn

```
and stmt =
  ...
  | PrintStack of expr
```

Result

```
dotnet run fac.c
[Vardec (TypI, "nFac"); Vardec (TypI, "resFac");
 Fundec
   (None, "main", [(TypI, "n")],
    Block
      [Dec (TypI, "i"); Stmt (Expr (Assign (AccVar "i", CstI 0)));
       Stmt (Expr (Assign (AccVar "nFac", CstI 0)));
       Stmt
         (While
            (Prim2 ("<", Access (AccVar "i"), Access (AccVar "n")),
             Block
               [Stmt
                  (Expr
                     (Assign
                        (AccVar "resFac", Call ("fac", [Access (AccVar "i")])))));
                Stmt
                  (Expr
```

```
                    (Assign
                       (AccVar "i", Prim2 ("+", Access (AccVar "i"), CstI 1))))]));
      Stmt (PrintStack (CstI 42))]);
Fundec
  (Some TypI, "fac", [(TypI, "n")],
   Block
     [Stmt
        (Expr
           (Assign (AccVar "nFac", Prim2 ("+", Access (AccVar "nFac"), CstI 1))));
      Stmt (PrintStack (Access (AccVar "nFac")));
      Stmt
        (If
           (Prim2 ("==", Access (AccVar "n"), CstI 0), Return (Some (CstI 1)),
            Return
              (Some
                 (Prim2
                    ("*", Access (AccVar "n"),
                     Call ("fac", [Prim2 ("-", Access (AccVar "n"), CstI 1)])))))))])]
```

Udvid machine samt comp

Machine.fs

```
type instr =
  ...
  | PRINTSTACK

let CODEPRINTSTACK   = 26;

let makelabenv (addr, labenv) instr =
    match instr with
    ...
    | PRINTSTACK     -> (addr+1, labenv)

let rec emitints getlab instr ints =
    match instr with
    ...
    | PRINTSTACK     -> CODEPRINTSTACK :: ints
```

Machine.java

Jeg valgte at tage en rekursiv approach til denne opgave da det gav lidt bedre mening når man skal opdatere en base peger, og holde styr på om det er globale eller lokale variable.

```
static int execcode(int[] p, int[] s, int[] iargs, boolean trace) {
  int bp = -999;     // Base pointer, for local variable access
```

```
  int sp = -1;  // Stack top pointer
  int pc = 0;       // Program counter: next instruction
  for (;;) {
    if (trace)
      printsppc(s, bp, sp, p, pc);
    switch (p[pc++]) {
     ...
    case PRINTSTACK:
      int N = s[sp--];
      System.out.println("-Print Stack "+N+"--------");
      printStack(s, bp, sp);
      break;
   ...
    default:
      throw new RuntimeException("Illegal instruction " + p[pc-1]
                                 + " at address " + (pc-1));
    }
  }
}

 static void printStack2(int[] s, int bp, int sp){
    if (bp == -999){
      System.out.println("Global");
      for (int i = sp; i>= 0; i--){
        printStackAddr(i);
        System.out.println(s[i]);
      }
    } else{
      System.out.println("Stack Frame");
      int i = sp;
      for (; i >= bp; i--){
        printStackAddr(i);
        printLocalTmp(s[i]);
      }
      printStackAddr(i);
      int newBp = s[i--];
      printBasePointer(newBp);
      printStackAddr(i);
      printRetAddr(s[i--]);

      printStack2(s, newBp, i);
    }
  }

  static void printStackAddr(int i){
    System.out.print("  s["+i+"]:  ");
```

```
  }
  static void printLocalTmp(int i){
    System.out.println("Local/Temp = "+i);
  }

  static void printBasePointer(int bp){
    System.out.println("bp = "+bp);
  }
  static void printRetAddr(int ret){
    System.out.println("ret = "+ret);
  }
```

Comp.fs

```
let rec cStmt stmt (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match stmt with
    ...
    | PrintStack(e) -> cExpr e varEnv funEnv @ [PRINTSTACK]
```

Jeg har følgende uddata fra at køre programmet i konsollen.

```
  opgave2 git:(main)  javac Machine.java && java Machine fac.out 1
-Print Stack 1--------
Stack Frame
  s[9]:  Local/Temp = 0
  s[8]:  bp = 4
  s[7]:  ret = 39
Stack Frame
  s[6]:  Local/Temp = 1
  s[5]:  Local/Temp = 0
  s[4]:  Local/Temp = 1
  s[3]:  bp = -999
  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 0.007 seconds
```

Min implementation er rekursiv, samt iterativ. Den er rekursiv når vi går fra een stak frame til den næste. Dog er den iterativ i den nuværende stak frame. Den printer alle værdier mens stak pegeren er større end eller lig basepegeren. Dette er da vi ved at base pegeren peger på førse værdi. Herefter printer vi base pegeren og retur pegeren. Hvorefter vi kan lave et rekursivt kald med den opdaterede stakpeger og base peger.

Dette fortsætter den med indtil basepegeren har værdien -999. Da vi ved at denne addresse er ugyldig så må de resterende værdier (efter retur pegeren) være globale variable.

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; // nFac = 0
    GOTO "L4"; //
Label "L3"; // while
    CSTI 1; //
    GETBP; CSTI 1; ADD; LDI; // put i on the top of the stack
    CALL (1,"L2"); STI; INCSP -1; // fac(i)
    GETBP; CSTI 1; ADD; // addressen for i
    GETBP; CSTI 1; ADD; LDI; // hent i
    CSTI 1; ADD; STI; INCSP -1; // i = i+1
    INCSP 0; // unødvendig instruktion
Label "L4"; // while check
    GETBP; CSTI 1; ADD; LDI; // i
    GETBP; CSTI 0; ADD; LDI; // n
    LT; IFNZRO "L3"; // if i < n goto L3
    CSTI 42; PRINTSTACK; // put 42 på stakken og kald PRINTSTACK
    INCSP -1; // fjern 42 fra stakken
    RET 0; // return fra main
Label "L2"; // fac
    CSTI 0; CSTI 0; LDI; CSTI 1; // putter nFac og 1 på stakken samt gør klar til at assigne
    ADD; STI; INCSP -1; //  udfører nFac = nFac + 1;
    CSTI 0; LDI; PRINTSTACK; // putter nFac på stakken og kalder printstack
    GETBP; CSTI 0; ADD; LDI; // hent n
    CSTI 0; EQ; IFZERO "L5"; // if (n != 0) goto l5
    CSTI 1; RET 1; GOTO "L6"; // return and clear 1 local value
Label "L5"; // else
```

```
    GETBP; CSTI 0; ADD; LDI; // hent n
    GETBP; CSTI 0; ADD; LDI; CSTI 1; SUB; // hent n - 1
    CALL (1,"L2"); // rekursivt kald
    MUL; RET 1; // n*fac(n-1)
Label "L6"; //
    INCSP 0; RET 0 // endeligt return så fac kan hoppe herned
```

Opgave 3

Udvid lexer, parser og absyn

Lexer er kun udvidet med eet nyt nøgleord, da resten af tokens er taget forbehold for.

```
let keyword s =
    match s with
    ...
    | "within"  -> WITHIN
    | _         -> NAME s
```

Parser er udvidet med en ny expression, samt den nye `within` token der har samme præcedens som $<$ og $>$, dette er så den binder hårdere end lig eller print, men ikke lige så hårdt som plus eller gange.

```
%token WITHIN

%right ASSIGN             /* lowest precedence */
%nonassoc PRINT
%left SEQOR
%left SEQAND
%left EQ NE
%left GT LT GE LE WITHIN
%left PLUS MINUS
%left TIMES DIV MOD
%nonassoc NOT AMP
%nonassoc LBRACK          /* highest precedence  */

ExprNotAccess:
    AtExprNotAccess                     { $1                  }
  ...
  | Expr WITHIN LBRACK Expr COMMA Expr RBRACK {Within($1, $4, $6)}
;
```

Til sidst er den abstrakte syntax blevet udvidet med en within expression. Denne tager 3 argumenter, som alle er expressions, da vi skal kunne tillade `x*2 within [y/3, z-1]`.

```
and expr =
```

```
  ...
  | Within of expr * expr * expr
```

Jeg får følgende syntax træ når jeg evaluerer programmet.

```
opgave3 git:(main)  dotnet run within.c
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",
                  Within
                    (Prim1 ("printi", CstI 42), Prim1 ("printi", CstI 40),
                     Prim1 ("printi", CstI 44)))))])]
```

Oversætter skema for within

```
E[e within [e1, e2]] =
    E[e]
    E[e1]
    LT
    NOT
    E[e2]
```

```
E[e]
LT
NOT
ADD
CSTI 2
EQ
```

Vi tjekker først om *e1* er mindre eller lig *e*, herefter tjekker vi om *e* er mindre eller lig *e2* herefter summerer vi op hvis resultatet er lig 2 så ved vi at begge er sande, eller ved vi et af udtrykkene er falske.

Implementer skemaet i Comp

Det er implementeret som i oversætter skemaet, *le*, *ge* og *res* afhænger af det samme miljø og kan derfor evalueres lige fra starten af, så sparer vi også en evaluering af *res*.

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match e with
    ...
    | Within(e, e1, e2) ->
      // Evaluate all expression
      let le = cExpr e1 varEnv funEnv
      let ge = cExpr e2 varEnv funEnv
      let res = cExpr e varEnv funEnv
      res
      @ le
      @ [LT; NOT] // if res is less than le then <= is not true
      @ ge
      @ res
      @ [LT; NOT; ADD; CSTI 2; EQ] // the same for ge, we add the results, which should be 2
```

Evaluering af det genererede bytekode, svaret er som forventet og stemmer overens med opgaven.

```
java Machine within.out
1 2 0 1 2 0 40 44 1 1 42 40 44 42 1
Ran 0.011 seconds
```