Programmer som data eksamen 2024

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

Denne besvarelse er udarbejdet af: Albert Ross Johannessen

Email: alrj@itu.dk

Student id: 21336

Opgave 1

Opgave 1.1

Icon fungere ved at en funktion kan have flere svar, hvis alle svar skal udskrives skal `Every` benyttes, som udfører det indre udtryk for alle resultater, da der ikke gøres det her, fås blot første resultat af enumerationen som i dette tilfælde er 1, da den går fra 1 til 10.

Opgave 1.2

Udtrykket som er givet ved

```
let examEx2 = Every(Write(Prim("+", CstI 4, FromTo(1, 10))))
printfn "%A" (examEx2 |> run)
```

som resulterer i

```
5 6 7 8 9 10 11 12 13 14 Int 0
```

Virker da `Prim("+",...,...)` lægger alle resultater sammen, da der kun er eet på venstresiden og flere på højresiden som resulterer i.

```
1+4 2+4 3+4 ...
```

Opgave 1.3

Udtrykket som er givet ved

```
let examEx3 =
    And(
        Every(Write(FromTo(1,5))),
        Every(Write(FromTo(6,10)))
    )
printfn "%A" (examEx3 |> run)
```

som resulterer i

```
1 2 3 4 5 6 7 8 9 10 Int 0
```

Giver det forventede da `And` først kører det første udtryk og herefter det andet udtryk, derfor kommer det i den korrekte rækkefølge.

Opgave 1.4

Der er blevet oprettet en ny expression type i Icon, som blot tager en int liste da generics ikke understøttes af expr typen.

```
type expr =
  ...
  | FromList of int list
```

Herefter er denne blevet håndteret i eval hvor der er lavet et rekursivt loop over elementerne til at listen er tom.

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    ...
    | FromList(lst) ->
      if List.length lst  = 0 then
        failwith "You cannot provide FromList with an empty list"
      else
        let rec loop lst =
          match lst with
          | [] -> econt ()
          | x::xs ->
            cont (Int x) (fun () -> loop xs)
        loop lst
```

Dette er blevet evalueret i en program.fs fil.

```
let examEx4 = Every(Write(FromList([1..5])))
printfn "%A" (examEx4 |> run)
```

Som giver følgende resultat

```
1 2 3 4 5 Int 0
```

Opgave 1.5

```
let examEx5 = And(
        Every(
            Write(FromList([1..5]))),
        Every(
            Write(FromList([6..10])))
    )
```

Opgave 1.6

Der er blevet oprettet en ny expression type i Icon, som tager 2 int lister som argument.

```
type expr =
  ...
  | FromMergeList of int list * int list
```

Herefter er denne blevet håndteret i eval hvor der er lavet et rekursivt loop over elementerne til at listerne er tomme. Dog er det vigtigt at sørge for at den skifter i mellem xs og ys listerne så derfor er der også en boolean der skifter mellem `true` og `false` hver iteration, afhængigt af om `x` skal printes eller `y`.

Hvis der ikke er flere elementer i een af listerne så bliver den ikke brugt da der blot skal tage det resterende som følge af beskrivelsen.

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    ...
    | FromMergeList(xs, ys) ->
      if List.length xs  = 0 || List.length ys = 0 then
        failwith "You cannot provide FromMergeList with an empty list"
      else
        let rec loop xs ys isXs =
          match xs, ys with
          | [], [] -> econt ()
          | x::xs, [] ->
            cont (Int x) (fun () -> loop xs ys true)
          | [], y::ys -> cont (Int y) (fun () -> loop xs ys true)
          | x::xs, y::ys ->
            if isXs then cont (Int x) (fun () -> loop xs (y::ys) false)
            else cont (Int y) (fun () -> loop (x::xs) ys true)
        loop xs ys true
```

Dette er blevet evalueret i en program.fs fil.

```
let examEx6 = Every(Write(FromMergeList([1..5],[6..12])))
printfn "%A" (examEx6 |> run)
```

Som giver følgende resultat

```
1 6 2 7 3 8 4 9 5 10 11 12 Int 0
```

Opgave 2

Opgave 2.1

FunPar er belevet udvidet med to nye tokens

```
%token NTH COMMA
```

og to nye udtryk i Expr

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

Expr:
    AtExpr     { $1  }
  ...
  | LPAR ExprLst RPAR   { Tup($2)    }
  | NTH LPAR Expr COMMA Expr RPAR    { Prim("nth", $3, $5)    }
;
```

hertil er der kommet en hjælpegrammer `ExprLst` til at lave listeudtryk så dette ikke behøves håndteres i Expr.

Det påkræves i `ExprLst` at der er mindst 2 elementer, ellers vil udtrykket fejle.

Opgave 2.2

FunLex er nu blevet udvidet til at lexe de nye tokens.

```
let keyword s =
    match s with
    ...
    | "nth"   -> NTH  (* Exam *)
    | _       -> NAME s

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

I Absyn er der tilføjet følgende linje

```
type expr =
  ...
  | Tup of expr list    (* Exam *)
```

Og nu gentager jeg lige FunPar for en sikkerhedsskyld, der kan læses yderligere om den i besvarelse *2.1*.

```
%token NTH COMMA

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

```
  | Expr COMMA ExprLst      { $1::$3}
;

Expr:
    AtExpr     { $1  }
  ...
  | LPAR ExprLst RPAR   { Tup($2)    }
  | NTH LPAR Expr COMMA Expr RPAR    { Prim("nth", $3, $5)    }
;
```

Det følgende udtryk kan nu parses via fsharp interactive.

```
fromString "let tup = (4,5,6) in nth(0,tup) + nth(1,tup) + nth(2,tup) end";;
```

som giver følgende resultat.

```
> fromString "let tup = (4,5,6) in nth(0,tup) + nth(1,tup) + nth(2,tup) end";;
val it: Absyn.expr =
  Let
    ("tup", Tup [CstI 4; CstI 5; CstI 6],
     Prim
       ("+",
        Prim
          ("+", Prim ("nth", CstI 0, Var "tup"),
           Prim ("nth", CstI 1, Var "tup")), Prim ("nth", CstI 2, Var "tup")))
```

Opgave 2.3

HigherFun er blevet udvidet typen value med en ny værditype, denne ræpresenterer en tupel værditype.

```
type value =
  ...
  | TupVal of value list (* Exam *)
```

Herefter er eval også blevet udvidet med den nyt primitive `nth` samt den nye expression type `Tup`.

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    ...
    | Prim(ope, e1, e2) ->
      let v1 = eval e1 env
      let v2 = eval e2 env
      match (ope, v1, v2) with
      ...
      | ("nth", Int i1, TupVal lst) -> lst[i1]
      |  _ -> failwith "unknown primitive or wrong type"
    ...
```

```
| Tup(lst) ->
    lst
    |> List.map (fun item -> eval item env)
    |> TupVal
```

Det er defineret i evalueringsregelerne at `nth` skal tage et udtryk der evaluerer til int som første værdi, og evaluerer til `TupVal` som anden værdi. Derfor fejler den hvis ikke det er de argumenter den får.

Da det er værdier af typen `value` der er i tuplen så kan argumentet hentes ud via indeksering.

Da der er oprette en ny `TupVal` værdi skal der blot mappes over listen givet i tup, med eval så det bliver omdannet til værdier. Denne nye liste kan indsættes ind i en `TupVal` og returnere.

Ved evaluering af

```
"let tup = (4,5,6) in nth(0,tup) + nth(1,tup) + nth(2,tup) end"
|> fromString
|> run;;
```

fås følgende resultat.

```
run(fromString "let tup = (4,5,6) in nth(0,tup) + nth(1,tup) + nth(2,tup) end");;
val it: HigherFun.value = Int 15
```

Dette er det forventede resultat, hvis 4, 5 og 6 lægges sammen giver det 15.

Opgave 3

Opgave 3.1

De 15 normale kald er givet af følgende årsager.

`main` kaldes, som det første, og det andet kald er til `fac`, hvilket nu er 2 kald. Nu kalder `fac` sig selv `n` gange uden at det er et halekald, grunden til dette er at der ganges `n` på det rekursive kald til `fac(n-1)`. Antallet af normale kald er nu oppe på 14. Det sidste kald kommer da `facT` kaldes. Altså er der 15 normale funktionskald.

Altså

$$1 + 1 + 12 + 1$$

De 12 halekald kommer udelukkende fra implementationen af `facT`, da den kalder sig selv `n` gange, hvor $n = 12$ i dette tilfælde.

`numCalls` og `numTailCalls` tæller ikke med da de er primitiver og ikke funktioner.

Opgave 3.2

Den abstrakte syntaks er blevet udvidet med den nye primitive type `Prim0`

```
and expr =
  ...
  | Prim0 of string (* Exam *)
  ...
```

Lexeren er udvidet med de to nye keywords som genererer tokens `NUMCALLS` samt `NUMTCALLS`.

```
let keyword s =
    match s with
    ...
    | "numCalls" -> NUMCALLS        (* Exam *)
    | "numTailCalls" -> NUMTCALLS   (* Exam *)
    | _          -> NAME s
```

Til sidst er de to nye tokens blevet tilføjet til parseren og evalueres til `Prim0`.

```
%token NUMCALLS NUMTCALLS

ExprNotAccess:
    AtExprNotAccess     { $1                  }
  ...
  | NUMCALLS            { Prim0("numCalls")   }
  | NUMTCALLS           { Prim0("numTailCalls")   }
;
```

Jeg ser ikke noget behov for at oprette præcedens regeler da de nye primitiver ikke kun skal ligge en værdi på stakken.

Nu kan *printstat.c* parse.

```
> fromFile "printstat.c"
Prog
  [Fundec
     (None, "main", [],
      Block
        [Stmt (Expr (Prim1 ("printi", Call ("fac", [CstI 12]))));
         Stmt (Expr (Prim1 ("printi", Call ("facT", [CstI 12; CstI 1]))));
         Stmt (Expr (Prim1 ("printi", Prim0 "numCalls")));
         Stmt (Expr (Prim1 ("printi", Prim0 "numTailCalls")))]);
   Fundec
     (Some TypI, "fac", [(TypI, "n")],
      Block
        [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)])))))]);
  Fundec
    (Some TypI, "facT", [(TypI, "n"); (TypI, "acc")],
     Block
       [Stmt
          (If
             (Prim2 ("==", Access (AccVar "n"), CstI 0),
              Return (Some (Access (AccVar "acc"))),
              Return
                (Some
                   (Call
                      ("facT",
                       [Prim2 ("-", Access (AccVar "n"), CstI 1);
                        Prim2 ("*", Access (AccVar "n"), Access (AccVar "acc"))])))))])]
```

Opgave 3.3

Java bytekode maskinen er blevet udvidet med to nye instruktioner, til dette formål er der også kommet en tæller værdi for antallet af kald samt halekald. Disse inkrementeres hver gang der er et kald eller halekald.

Ved `NUMCALLS` og `NUMTCALLS` inkrementeres stak pegeren med én og herefter lægges hhv. `numCalls` eller `numTCalls` på toppen af stakken.

```
final static int
    ..., NUMCALLS = 26, NUMTCALLS = 27;

  static int execcode(int[] p, int[] s, int[] iargs, boolean trace) {
    int numCalls = 0;   // Exam
    int numTCalls = 0;  // Exam
    ...
    for (;;) {
      if (trace)
        printsppc(s, bp, sp, p, pc);
      switch (p[pc++]) {
      ...
      case CALL: {
        numCalls++;     // Exam
        ...
      } break;
      case TCALL: {
        numTCalls++;    // Exam
        ...
```

```
      } break;
      ...
      case NUMCALLS:{   // Exam
        s[++sp] = numCalls;
      }break;
      case NUMTCALLS:{  // Exam
        s[++sp] = numTCalls;
      }break;
      ...
      default:
        throw new RuntimeException("Illegal instruction " + p[pc-1]
                                   + " at address " + (pc-1));
      }
    }
  }
```

Opgave 3.4

Det er ikke beskrevet at der skal tilføjes nye bytekode instruktioner, til compileren, men da det påkræves for at programmet kan omdannes til bytekode, har jeg valgt at udvide *Machine.fs* med de 2 nye instruktioner.

Først skal Machine.fs udvides for at understøtte de to nye instruktioner.

```
type instr =
  ...
  | NUMCALLS     (* Exam *)
  | NUMTCALLS    (* Exam *)

let CODECSTI   = 0
...
let CODENUMCALLS = 26    (* Exam *)
let CODENUMTCALLS = 27   (* Exam *)

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

let rec emitints getlab instr ints =
    match instr with
    ...
    | NUMCALLS -> CODENUMCALLS :: ints
    | NUMTCALLS -> CODENUMTCALLS :: ints
```

Hverken af dem har nogle argumenter i bytekoden så derfor skal addressen i

`makelabenv` blot inkrementeres med 1 og koderne skal appendes på det resterende program i `emitints`.

Nu hvor de nye instruktioner er tilføjet kan der nu understøttes den nye expression type `Prim0`.

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) (C : instr list) : instr list =
    match e with
    ...
    | Prim0(ope) ->
      match ope with
      | "numCalls" -> NUMCALLS :: C
      | "numTailCalls" -> NUMTCALLS :: C
      | _ -> failwith "unknown primitive 0"
    ...
```

`Prim0` appender blot instruktionen på resten af programmet, og fejler hvis primitiven er ukendt.

Der kommer følgende uddata når *printstat.c* compiles.

Den abstrakte syntaks.

```
Prog
  [Fundec
     (None, "main", [],
      Block
        [Stmt (Expr (Prim1 ("printi", Call ("fac", [CstI 12]))));
         Stmt (Expr (Prim1 ("printi", Call ("facT", [CstI 12; CstI 1]))));
         Stmt (Expr (Prim1 ("printi", Prim0 "numCalls")));
         Stmt (Expr (Prim1 ("printi", Prim0 "numTailCalls")))]);
   Fundec
     (Some TypI, "fac", [(TypI, "n")],
      Block
        [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)])))))))]);
   Fundec
     (Some TypI, "facT", [(TypI, "n"); (TypI, "acc")],
      Block
        [Stmt
           (If
              (Prim2 ("==", Access (AccVar "n"), CstI 0),
               Return (Some (Access (AccVar "acc"))),
```

```
                Return
                  (Some
                     (Call
                        ("facT",
                         [Prim2 ("-", Access (AccVar "n"), CstI 1);
                          Prim2 ("*", Access (AccVar "n"), Access (AccVar "acc"))])))))])]
```

Den resulterende bytekode.

```
[
    LDARGS; CALL (0, "L1"); STOP; Label "L1"; CSTI 12; CALL (1, "L2");
    PRINTI; INCSP -1; CSTI 12; CSTI 1; CALL (2, "L3"); PRINTI; INCSP -1;
    NUMCALLS; PRINTI; INCSP -1; NUMTCALLS; PRINTI; RET 0; Label "L2"; GETBP;
    LDI; IFNZRO "L4"; CSTI 1; RET 1; Label "L4"; GETBP; LDI; GETBP; LDI;
    CSTI 1; SUB; CALL (1, "L2"); MUL; RET 1; Label "L3"; GETBP; LDI;
    IFNZRO "L5"; GETBP; CSTI 1; ADD; LDI; RET 2; Label "L5"; GETBP; LDI;
    CSTI 1; SUB; GETBP; LDI; GETBP; CSTI 1; ADD; LDI; MUL; TCALL (2, 2, "L3")
]
```

Evalueringen af *Machine.java*

```
java Machine printstat.out
479001600 479001600 15 12
Ran 0.01 seconds
```

Opgave 4

Opgave 4.1

```
LDARGS;                 // Load parameter fra kommadolinie
                        // Der er ingen
CALL (0, "L1");         // Kald main
STOP;                   // stop ved retur fra main
Label "L1";             // main
    CSTI 12; CALL (1, "L2"); PRINTI; // kald fac med 12 og print resultat
    INCSP -1;           // fjern resultatet fra stakken
    CSTI 12; CSTI 1; CALL (2, "L3"); PRINTI;  // kald facT og print resultat
    INCSP -1;           // fjern resultat fra stakken
    NUMCALLS; PRINTI;   // print antallet af normale kald
    INCSP -1;           // fjern antallet af normale kald fra stakken
    NUMTCALLS; PRINTI;  // print antallet af halekald
    RET 0;              // returner og fjern 0 funktions argument
Label "L2";             // fac
    GETBP; LDI; IFNZRO "L4"; // hent første argument hvis lig nul returner
    CSTI 1; RET 1;      // returner 1 hvis ovenstående er falsk
  Label "L4";           // else
    GETBP; LDI;         // hent første argument på stakken
```

```
    GETBP; LDI; CSTI 1; SUB; // hent første argument på stakken
                             // og træk 1 fra; n-1
    CALL (1, "L2");      // rekursivt kald til fac med n-1
    MUL;                 // gang resultatet fra kaldet med n
    RET 1;               // returner og fjern det første argument
Label "L3";              // facT
    GETBP; LDI; IFNZRO "L5"; // hent første argument hvis lig 0 returner acc
    GETBP; CSTI 1; ADD; LDI; RET 2; // andet argument (acc) hentes
                             // og returnerer værdien herefter fjernes
                             // de to argumenter fra stakken
  Label "L5";            // else
    GETBP; LDI; CSTI 1; SUB; // læg n-1 på stakken
    GETBP; LDI;          // læg n på stakken
    GETBP; CSTI 1; ADD; LDI; // læg acc på stakken
    MUL;                 // gang stakken to øverste værdier; n * acc
    TCALL (2, 2, "L3")   // rekursivt halekald med 2 argumenter; n-1 samt n*acc
```

Opgave 4.2

```
24      // LDARGS
19 0 5  // CALL(0, 5)
25      // STOP
0 12    // CSTI 12
19 1 31 // CALL(1, 31)
22      // PRINTI
15 -1   // INCSP -1
```