BSPRDAT1KU **IT University, E2022**

Skriftlig eksamen, Programmer som Data
Januar 2023

Version 1.00 af January 5, 2023

Dette eksamenssæt har 10 sider. Tjek med det samme at du har alle siderne.
Eksamenssættet udleveres elektronisk fra kursets hjemmeside mandag den 9. januar 2023 kl 08:00.
Besvarelsen skal afleveres elektronisk i LearnIt senest **tirsdag den 10. januar 2023 kl 14:00** som følger:

- Besvarelsen skal uploades på kursets hjemmeside i LearnIt under **Ordinary exam assignment**.
- Der kan uploades en fil, som skal have en af følgende typer: `.txt` eller `.pdf`. Hvis du for eksempel laver besvarelsen i LaTeX eller Word, så generer en pdf-fil. Hvis du laver en tegning i hånden, så scan den og inkluder det skannede billede i det dokument du afleverer.

Der er 4 opgaver. For at få fulde point skal du besvare alle opgaverne tilfredsstillende.

Hvis der er uklarheder, inkonsistenser eller tilsyneladende fejl i denne opgavetekst, så skal du i din besvarelse beskrive disse og beskrive hvilken tolkning af opgaveteksten du har anvendt ved besvarelsen. Hvis du mener det er nødvendigt at kontakte opgavestiller, så send en email til `sap@itu.dk` med forklaring og angivelse af problem i opgaveteksten.

Din besvarelse skal laves af dig og kun dig, og det gælder både programkode, lexer- og parserspecifikationer, eksempler, osv., og den forklarende tekst der besvarer opgavespørgsmålene. Det er altså ikke tilladt at lave gruppearbejde om eksamen.

Din besvarelse skal indeholde følgende erklæring:

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

Du må bruge alle bøger, forelæsningsnoter, forelæsningsplancher, opgavesæt, dine egne opgavebesvarelser, internetressourcer, lommeregnere, computere, og så videre.

Du må **naturligvis ikke plagiere** fra andre kilder i din besvarelse, altså forsøge at tage kredit for arbejde, som ikke er dit eget. Din besvarelse må ikke indeholde tekst, programkode, figurer, tabeller eller lignende som er skabt af andre end dig selv, med mindre der er fyldestgørende kildeangivelse, dvs. at du beskriver oprindelsen af den pågældende tekst (eller lignende) på en komplet og retvisende måde. Det gælder også hvis den inkluderede kopi ikke er identisk, men tilpasset fra tekst eller programkode fra lærebøger eller fra andre kilder.

Hvis en opgave kræver, at du definerer en bestemt funktion, så må du gerne **definere alle de hjælpefunktioner du vil**, men du skal definere funktionen så den har den ønskede type og giver det ønskede resultat.

Udformning af besvarelsen

Besvarelsen skal bestå af forklarende tekst (på dansk eller engelsk) der besvarer spørgsmålene, med væsentlige programfragmenter indsat i den forklarende tekst, eller vedlagt i bilag (der klart angiver hvilke kodestumper der hører til hvilke opgaver).

Vær omhyggelig med at programfragmenterne beholder det korrekte layout når de indsættes i den løbende tekst, for F#-kode er som bekendt layoutsensitiv.

Det forventes at løsninger suppleres med tests der demonstrerer at løsningerne fungerer efter hensigten.

Snydtjek

Til denne eksamen anvendes *Snydtjek*. Cirka 20% vil blive udtrukket af studieadministrationen i løbet af eksamen. Navne bliver offentliggjort på kursets hjemmeside tirsdag den 10. januar klokken 14:00. Disse personer skal møde op i det Zoom møde, som også opslås på kursets hjemmeside, sammen med de udtrukne navne. Man vil blive trukket ind til mødet en af gangen.

Til Snydtjek er processen, at hver enkelt kommer ind i 5 minutter, hvor der stilles nogle korte spørgsmål omkring den netop afleverede besvarelse. Formålet er udelukkende at sikre at den afleverede løsning er udfærdiget af den person, som har uploaded løsningen. Du skal huske dit studiekort.

Det er obligatorisk at møde op til snydtjek i tilfælde af at du er udtrukket. Udeblivelse medfører at eksamensbesvarelsen ikke er gyldig og kurset ikke bestået. Er man ikke udtrukket skal man ikke møde op.

Opgave 1 (25%) Icon

Kapitel 11 i *Programming Language Concepts* (PLC) introducerer "continuations" og "back tracking" samt sproget Icon. Koden der anvendes her findes i kursets git-repository, `Lectures/Lec11/Cont/Icon.fs`.

I Icon kan vi f.eks. skrive `5 to 12`, som er en generator for tallene mellem 5 og 12, begge inklusive. Ved at anvende implementationen i filen `Icon.fs` kan vi udtrykke dette i abstrakt syntaks:

```
let numbers = FromTo(5,12)
```

og køre eksemplet, der giver tallet 5 som resultat, inde fra F# fortolkeren:

```
> run numbers;;
val it : value = Int 5
```

1. Skriv et Icon udtryk, som udskriver værdierne 5 6 7 8 9 10 11 12 på skærmen, fx.:

   ```
   > run ...;;
   5 6 7 8 9 10 11 12 val it : value = Int 0
   ```

   hvor `...` repræsenterer din løsning. Generatoren `numbers` skal indgå i din løsning. Forklar hvorledes du får udskrevet alle 8 tal.

2. Skriv et Icon udtryk, som udskriver alle tal genereret af `numbers`, som er større end 10.

   ```
   > run ...;;
   11 12 val it : value = Int 0
   ```

   hvor `...` repræsenterer din løsning. Generatoren `numbers` skal indgå i din løsning. Du skal forklare hvordan din løsning fungerer.

3. Skriv et Icon udtryk, som udskriver nedenstående linier.

   ```
   > run ...;;

    6 7 8 9 10 11 12
    7 8 9 10 11 12
    8 9 10 11 12
    9 10 11 12
    10 11 12
    11 12
    12
    val it : value = Int 0
   ```

   hvor `...` repræsenterer din løsning. Generatoren `numbers` skal indgå i din løsning. Du skal forklare hvordan din løsning fungerer.

4. Udvid implementationen af Icon med en ny generator `FromToChar(`c_1`,`c_2`)`, som genererer tegnene fra c_1 til c_2, begge inklusive. Det antages, at tegnet c_1 ligger før c_2 i tegntabellen. Generatoren `FromToChar` fejler med det samme, hvis c_1 ligger efter c_2 i tegntabellen.

   Lad generatoren `chars` være defineret ved

   ```
   let chars = FromToChar('C','L')
   ```

   Et par eksempler nedenfor:

   ```
   > run chars;;
   val it : value = Str "C"

   > run (Every(Write(chars)));;
   C D E F G H I J K L val it : value = Int 0
   ```

Bemærk, at generatoren `chars` genererer strenge, f.eks. `Str "C"` ovenfor.

5. Udvid implementationen af Icon, således at primitivet `Prim("<",`e_1`,` e_2`)` også fungerer, når e_1 og e_2 er strenge. Eksempelvis

```
> run (Prim("<", CstS "A", CstS "B"));;
val it : value = Str "B"

> run (Prim("<", CstS "B", CstS "A"));;
Failed
val it : value = Int 0
```

   I det første eksempel er `"A"` mindre end `"B"` og resultatet er `"B"`. Det andet eksempel fejler fordi `"B"` ikke er mindre end `"A"`. Demonstrer at din løsning fungerer med ovenstående eksempler.

6. Skriv et Icon udtryk, som udskriver nedenstående linier.

```
> run ...;;
H I J K L val it : value = Int 0
```

   hvor `...` repræsenterer din løsning. Generatoren `chars` skal indgå i din løsning. Du skal forklare hvordan din løsning fungerer.

Opgave 2 (25%) List–C: Stack

I denne opgave implementerer vi stakke (eng. *stacks*), således at de allokeres på hoben (eng. *heap*). Sproget list–C er beskrevet i afsnit 10.7 i PLC. Koden der anvendes her findes i kataloget `Lectures/Lec10/ListC` i kursets git repository. Spildopsamling (eng. *garbage collection*) behøver ikke at fungere for at løse denne opgave.

Stakke, som defineret her, kan indeholde et foruddefineret maksimum antal elementer. Figuren nedenfor angiver hvordan vi vil repræsentere en stak med plads til maksimum N elementer $v_1, \ldots, v_N$ på hoben:

	p	$p+1$	$p+2$	$p+3$		$p+N+1$	$p+N+2$	
...	*header*	N	*top*	v_1	v_2	...	v_N	...

Første element efter *header*, adresse $p+1$, indeholder det maksimale antal elementer stakken kan have, N. På adresse $p+2$ findes antal elementer i stakken, *top*, som er et tal mellem 0 og N. Hvis stakken er tom er *top* lig 0. Adresse på øverste element, i en ikke tom stak, er $p+2+$ *top*. Afsnit 10.7.3 og 10.7.4 i PLC (se `version for 64 bit` i Lektion 10) beskriver indholdet af *header*, der bl.a. indeholder et tag og størrelsen af blokken som antal ord (*eng.* words). En stak fylder $N+3$ ord på hoben. Som eksempel, er stakken med maksimal størrelse på 3 med elementerne `42` og `43` repræsenteret nedenfor, startende på adresse 112 i hoben. Adresse 114 indeholder *top*, som er 2, fordi stakken indeholder 2 elementer med 44 øverst på stakken. Adresse 117 har plads til et enkelt element mere på stakken.

	112	113	114	115	116	117	
...	*header*	`3`	`2`	`43`	`44`	_	...

Vi udvider list–C med fire nye funktioner:

- `createStack(`N`)` allokerer en stak af størrelse N på hoben. Alle elementer i stakken initialiseres til værdien 0. Hvis $N < 0$ skal programmet stoppe med passende fejlmeddelelse.
- `pushStack(`s`,`v`)` lægger værdien v øverst på stakken s og lægger en til *top*. Programmet skal stoppe med passende fejlmeddelelse, hvis der ikke er plads på stakken.
- `popStack(`s`)` returnerer den øverste værdi på stakken. Den øverste værdi fjernes og *top* reduceres med en. Programmet skal stoppe med passende fejlmeddelelse, hvis stakken allerede er tom.
- `printStack(`s`)` udskriver stakken s på skærmen.

Vi anvender typen `dynamic` til at repræsentere stakke, se eksempel `stack.lc` nedenfor:

```
void main() {
  dynamic s;
  s = createStack(3);
  pushStack(s,42);
  pushStack(s,43);
  printStack(s);

  print popStack(s);
  print popStack(s);
  printStack(s);
}
```

Opgaven er at implementere stakke således, at ovenstående program kan køre.

1. Den abstrakte syntaks `Absyn.fs` indeholder mulighed for primitiver med 1 (`Prim1`) og 2 (`Prim2`) argumenter. Du kan dermed repræsentere alle 4 funktioner ovenfor ved hjælp af de to primitiver.

   Udvid lexer– og parser–specifikationen således at ovenstående funktioner kan parses og generer et passende abstrakt syntakstræ, eksempelvis for `stack.lc`:

```
% listcc.exe stack.lc
List-C compiler v 1.0.0.0 of 2012-02-13
Compiling stack.lc to stack.out
Prog
  [Fundec
     (None, "main", [],
      Block
        [Dec (TypD, "s");
         Stmt (Expr (Assign (AccVar "s", Prim1 ("createStack", CstI 3))));
         Stmt (Expr (Prim2 ("pushStack", Access (AccVar "s"), CstI 42)));
         Stmt (Expr (Prim2 ("pushStack", Access (AccVar "s"), CstI 43)));
         Stmt (Expr (Prim1 ("printStack", Access (AccVar "s"))));
         Stmt (Expr (Prim1 ("printi", Prim1 ("popStack", Access (AccVar "s")))));
         Stmt (Expr (Prim1 ("printi", Prim1 ("popStack", Access (AccVar "t")))));
         Stmt (Expr (Prim1 ("printStack", Access (AccVar "s"))))])]
```

Hint: For at printe det abstrakte syntakstræ kan du ændre funktionen `fromFile` i `Parse.fs` således at træet udskrives inden det returneres.

Hint: Du kan oprette en token for hver af de fire funktioner ovenfor, som lexeren returner ved at matche ud på funktionsnavnene i lexer funktionen `keyword`.

2. Tilføj fire nye bytekode instruktioner `CREATESTACK`, `PUSHSTACK`, `POPSTACK` og `PRINTSTACK` for at "allokere", "pushe", "poppe" og "udskrive" stakke.

	Instruction	Stack before		Stack after	Effect
0	`CSTI` i	s	$\Rightarrow$	s, i	Push constant i
...					
32	`CREATESTACK`	s, N	$\Rightarrow$	s, p	Create stack of size N in the heap. Put pointer p to the created stack on the stack.
33	`PUSHSTACK`	s, p, v	$\Rightarrow$	s, p	Push value v on stack pointed at by p. The pointer p is left on the stack.
34	`POPSTACK`	s, p	$\Rightarrow$	s, v	The top value v on stack pointed at by p is left on the stack.
35	`PRINTSTACK`	s, p	$\Rightarrow$	s, p	The stack p is printed on the console. Pointer p to stack is left on stack.

I tabellen ovenfor er den eksisterende bytekode instruktion 0 `CSTI` medtaget til sammenligning. Du kan benytte tagget 1 for stakke.

Udvid `Machine.fs` og `listmashine.c` med de fire bytekode instruktioner. I `listmachine.c` skal du anvende `allocate` ved implementationen af `CREATESTACK`. Derudover skal du tagge (`Tag`) og untagge (`Untag`) de skalare værdier N og *top*. Ved `CREATESTACK` skal du også initialisere den afsatte plads til `Tag(0)`. Husk at håndtere fejlsituationer, som beskrevet ovenfor, f.eks. at "poppe" en tom stak.

Bytekode instruktionen `PRINTSTACK` p, hvor p peger på en stak, skal printes således: `STACK(`N`,`*top*`) [`v_1 ... v_{top}`]`. En tom stak printes `STACK(`N`,0) []`.

3. Udvid `Comp.fs` til at generere kode for de fire nye funktioner.

   Kør ovenstående eksempelprogram `stack.lc` og vis uddata, eksempelvis.

```
% ./ListVM/ListVM/listmachine stack.out
STACK(3, 2)=[ 42 43 ]
43 42 STACK(3, 0)=[ ]
```

4. Beskriv de vigtigste test cases man bør lave for at teste `createStack` og `pushStack`. Implementer mindst to test cases for hver af de to bytekode instruktioner og vis resultatet af at køre testene.

Vis (i udklip) de modifikationer du har lavet til filerne `Absyn.fs`, `CLex.fsl`, `CPar.fsy`, `listmachine.c` og `Comp.fs` for at implementere stakke. Giv en skriftlig forklaring på modifikationerne.

Opgave 3 (25%) Micro–C: Tupler

I denne opgave udvider vi sproget micro–C, som beskrevet i kapitel 7 og 8 i PLC, med tupler allokeret på stakken (eng. *stack*). Koden der anvendes her findes i kataloget `Lectures/Lec06/MicroC` i kursets git repository. Tupler kan have 1, 2 eller flere elementer. Figuren nedenfor angiver hvordan et tuppel med N elementer, $v_1, \ldots, v_N$, repræsenteres på stakken. Stakken vokser fra venstre mod højre, og p angiver adresse for første element i tuplen.

	p	$p+1$		$p+N$	
...	v_1	v_2	...	v_N	...

Et tuppels størrelse kendes på oversætter tidspunkt og kan dermed allokeres på stakken. Som eksempel er tuplen med elementerne `55`, `56` og `57` repræsenteret nedenfor.

	p	$p+1$	$p+2$	
...	55	56	57	...

Opgaven er at udvide micro–C således at vi kan erklære variable af en ny tuppel type. Derudover kan vi opdatere og læse fra tupler efter samme principper, som for tabeller (eng. *"arrays"*).

Hvor vi for tabeller anvender kantede parenteser, `[ ]`, anvender vi paranteser og lodret streg (eng. *"vertical bar"*) `(| |)` for tupler. I eksemplet, `tupple.c`, nedenfor erklæres `t1` til at være en tuppel med 2 elementer af typen `int`. Tuplen opdateres ved almindelig tildeling (eng. *"assignment"*), `t1(|0|) = 55;`. Første element har indeks 0.

```
void main() {
  int t1(|2|);
  t1(|0|) = 55;
  print t1(|0|); // 55
  t1(|1|) = 56;
  print t1(|1|); // 56
  int i;
  i = 0;
  while (i < 2) {
    print t1(|i|); // 55 56
    i = i + 1;
  }
}
```

Tilsvarende tabeller, er der ikke noget check af at indekset i, `t(|`i`|)`, ind i tuplen t er positiv, eller går ud over antallet af elementer i tuplen t. Således kan der nemt laves programmer med uforudsigelig effekt. Opgaven er at implementere tupler, således at ovenstående program kan afvikles.

1. **Absyn.fs**: Udvid typen `typ` med en ny tuppel type `TypT`, som er identisk med typen for tabeller `TypA`. En tuppel med 2 integer elementer repræsenteres således: `TypT (TypI, Some 2)`. En tuppel med ukendt antal integer elementer repræsenteres således: `TypT (TypI, None)`. Vis, at du får samme resultat som nedenstående med din implementation af `TypT`:

   ```
   > open Absyn;;
   > TypT (TypI, Some 2);;
   val it : typ = TypT (TypI, Some 2)

   > TypT (TypI, None);;
   val it : typ = TypT (TypI, None)
   ```

   Tilsvarende tabeller, skal vi kunne tilgå adressen (eng. *"L-value")* på hvert element i tuplen. Til dette udvider vi typen `access` med en ny konstruktør `TupIndex`, der repræsenterer addressen på et element indekseret ind i tuplen; `TupIndex` er magen til `AccIndex` for tabeller. Eksempelvis vil en tuppel t indekseret ind til element e, `t(|`e`|)`, være repræsenteret således: `TupIndex (AccVar t, e)`, hvor t er en variabel og e et udtryk der evaluerer til en integer (indeks). Vis, at du får samme resultat som nedenstående med din implementation af `TupIndex`, `t1(|0|)`:

```
> open Absyn;;
> TupIndex (AccVar "t1", CstI 0);;
val it : access = TupIndex (AccVar "t1", CstI 0)
```

2. **CLex.fsl**: Udvid lexer med genkendelse af to nye tokens:

Regulært udtryk	Token
(|	LPARBAR
|)	BARRPAR

3. **CPar.fsy**: Tilføj de to nye tokens `LPARBAR` og `BARRPAR`.

   For grammatikreglen `Vardesc` skal du parse erklæring af tupler på samme måde som tabeller. Den eneste forskel er, at hvor tabeller genkendes med `LBRACK` og `RBRACK`, skal du benytte `LPARBAR` og `BARRPAR` for tuppel typen og i stedet for `TypA` anvende `TypT`. Du kan læse mere om at parse erklæringer i afsnit 7.6.4 i PLC.

   Hint: For at undgå shift/reduce fejl, skal du gøre `LPARBAR` non–associativ, `%nonassoc`.

   Vis, at du kan bygge lexer og parser samt får samme resultat som nedenstående eksempler:

```
> open ParseAndComp;;
> fromString "void main() {int t1(|2|);}";;
val it : Absyn.program =
  Prog [Fundec (None, "main", [], Block [Dec (TypT (TypI, Some 2), "t1")])]

> fromString "void main() {int t1(||);}";;
val it : Absyn.program =
  Prog [Fundec (None, "main", [], Block [Dec (TypT (TypI, None), "t1")])]
```

   For at parse t`(|`e`|)` som *L–værdi*, skal du udvide grammatikreglen `Access`, så t`(|`e`|)` parses til `TupIndex(`t`,`e`)`. Dette svarer præcist til tabeller, hvor a`[`e`]` parses til `AccIndex(`a`,`e`)`.

   Vis, at du kan bygge lexer og parser samt får samme resultat som nedenstående eksempler:

```
> fromString "void main() {t1(|0|) = 55;}";;
val it : Absyn.program =
  Prog
    [Fundec
       (None, "main", [],
        Block [Stmt (Expr (Assign (TupIndex (AccVar "t1", CstI 0), CstI 55)))])]

> fromString "void main() {print t1(|0|);}";;
val it : Absyn.program =
  Prog
    [Fundec
       (None, "main", [],
        Block
          [Stmt
             (Expr (Prim1 ("printi", Access (TupIndex (AccVar "t1", CstI 0)))))])]
```

4. **Comp.fs**: Du skal implementere erkæring af tupler samt adgang til *L–værdien* af et element i tuplen.

   - For erklæring skal du implementere tilfældet for `TypT (_, Some` i`)` i funktionen `allocate`, der har til formål at allokere tuplen på stakken. Implementationen ligner den for variable bortset fra at der skal allokeres plads til i elementer og ikke blot et element. Som for tabeller understøtter tupler kun elementer der fylder et ord (eng. *"word"*). Eksempelvis understøttes tupler i tupler ikke.
   - For *L–værdi* skal du udvide funktionen `cAccess` med `TupIndex(`*acc*`,`*idx*`)`. Implementationen af `TupIndex` ligner implementationen af `AccIndex` bortset fra at tabeller er repræsenteret en smule anderledes end tupler. For tupler repræsenterer *acc* adressen på første element hvorefter indekset *idx* adderes. For tabeller repræsenterer *acc* en peger til første element i tabellen, se afsnit 8.8 i PLC.

5. Vis (i udklip) de modifikationer du har lavet til filerne `Absyn.fs`, `CLex.fsl`, `CPar.fsy` og `Comp.fs` for at implementere tupler. Giv en skriftlig forklaring på modifikationerne.

6. Dokumenter, ved at oversætte og køre ovenstående eksempelprogram, `tupple.c`, at du får forventet uddata.

```
% java Machine tupple.out
55 56 55 56
Ran 0.015 seconds
```

Opgave 4 (25%) Micro-ML: Lists

Kapitel 5 i *Programming Language Concepts* (PLC) introducerer evaluering af et højereordens funktionssprog og kapitel 6 introducerer polymorf typeinferens. Koden der anvendes her findes i kataloget `Lectures/Lec05/Fun` i kursets git repository. Opgaven er at udvide funktionssproget med lister (eng. *lists*), således at vi kan evaluere udtryk der manipulerer lister, se eksempel `ex01` nedenfor:

```
let l1 = [2, 3] in
  let l2 = [1, 4] in
    l1 @ l2 = [2, 3, 1, 4]
  end
end
```

Vi har tilføjet to syntaktiske konstruktioner:

- En ikke tom liste: $[e_1, \ldots, e_n], n \geq 1$.
- En operator e_1 `@` e_2 som sætter to lister sammen repræsenteret ved udtrykkene e_1 og e_2. Operatoren `@` har samme præcedens og associativitet som operatoren `+`.

Derudover kan vi sammenligne to lister med den eksiserende oprator, `=`.

1. Du skal udvide lexer `FunLex.fsl` og parser `FunPar.fsy` med support for lister og `@` operatoren defineret ovenfor. Den abstrakte syntaks i `Absyn.fs` er udvidet med `List` der repræsenterer et listeudtryk.

   ```
   type expr =
     | CstI of int
     | CstB of bool
     | List of expr list (* Exam, E2022 *)
     ...
   ```

   Du ser den abstrakte syntaks for ovenstående eksempel `ex01` nedenfor:

   ```
   > fromString @"let l1 = [2, 3] in
   -     let l2 = [1, 4] in
   -       l1 @ l2 = [2, 3, 1, 4]
   -     end
   -   end
   - ";;
   val it : Absyn.expr =
     Let ("l1", List [CstI 2; CstI 3],
       Let ("l2", List [CstI 1; CstI 4],
         Prim ("=",
               Prim ("@", Var "l1", Var "l2"),
               List [CstI 2; CstI 3; CstI 1; CstI 4])))
   ```

   Vis dine tilføjelser og at din løsning giver tilsvarende resultat for `ex01`. Vis ydermere resultatet af at parse nedenstående eksempler:

Reference	Example
ex02	`let l = [] in l end`
ex03	`let l = [43] in l @ [3+4] end`
ex04	`let l = [3] in l @ [3] = [3+4] end`
ex05	`let f x = x+1 in [f] end`
ex06	`let id x = x in [id] end`

   Bemærk, at tomme lister ikke understøttes, hvorfor `ex02` skal fejle.

2. Figur 4.3 på side 65 i PLC viser evalueringsregler for micro–ML, som vi har udvidet med lister. Nedenfor ses evalueringsregler for de nye konstruktioner:

$$(\textit{list})\ \frac{\rho \vdash e_i \Rightarrow v_i,\ 1 \leq i \leq n}{\rho \vdash [e_1, \ldots, e_n] \Rightarrow \texttt{ListV}[v_1, \ldots, v_n]}$$

$$(@)\frac{\rho \vdash e_1 \Rightarrow \texttt{ListV}[v_1,\ldots,v_n] \quad \rho \vdash e_2 \Rightarrow \texttt{ListV}[\,w_1,\ldots,w_m]}{\rho \vdash e_1 @ e_2 \Rightarrow \texttt{ListV}[v_1,\ldots,v_n,w_1,\ldots,w_m]}$$

$$(=T)\frac{\rho \vdash e_1 \Rightarrow \texttt{ListV}[v_1,\ldots,v_n] \quad \rho \vdash e_2 \Rightarrow \texttt{ListV}[\,w_1,\ldots,w_n] \quad v_i = w_i,\ 1 \le i \le n}{\rho \vdash e_1 = e_2 \Rightarrow \texttt{true}}$$

$$(=F)\frac{\rho \vdash e_1 \Rightarrow \texttt{ListV}[v_1,\ldots,v_n] \quad \rho \vdash e_2 \Rightarrow \texttt{ListV}[\,w_1,\ldots,w_m] \quad n \neq m \ \text{ or } \ \exists i : v_i \neq w_i}{\rho \vdash e_1 = e_2 \Rightarrow \texttt{false}}$$

Reglen *list* genererer en liste som repræsenteres ved $\texttt{ListV}[v_1,\ldots,v_n]$. Når lister sættes sammen, kommer elementerne i e_2 efter elementerne i e_1, regel @. To lister er ens, regel *=T*, hvis de har samme antal elementer, elementerne er ens og står i samme rækkefølge; ellers er listerne ikke ens, regel *=F*.

Udvid typen `value` og funktionen `eval` i `HigherFun.fs`, således at udtryk med lister kan evalueres som defineret af reglerne ovenfor. Vi repræsenterer lister med den indbyggede `list`–type i F#, som både understøtter sammensætning af lister, `@`, og lighed =.

```
type value =
  | Int of int
  | Closure of string * string * expr * value env (* (f, x, fBody, fDeclEnv) *)
  | ListV of value list       (* Exam, E2022 *)
```

Vis dine tilføjelser og resultatet af at evaluere eksemplerne `ex01`, ... `ex06`. Eksempelvis for `ex06`:

```
> open ParseAndRunHigher;;
> run(fromString "let id x = x in [id] end ");;
val it : HigherFun.value = ListV [Closure ("id", "x", Var "x", [])]
```

3. Figur 6.1 på side 102 i PLC viser typeregler for micro–ML. Vi antager, at vi har en type t `list` der repræsenterer en liste med elementer af type t. Nedenfor er typereglerne for *list*, = og `@` lavet. Reglen for *list* kræver, at alle elementer e_i i listen har samme type t.

$$(list)\frac{\rho \vdash e_i : t,\ 1 \le i \le n}{\rho \vdash [e_1,\ldots,e_n] : t\ \texttt{list}} \qquad (@)\frac{\rho \vdash e_1 : t\ \texttt{list} \quad \rho \vdash e_2 : t\ \texttt{list}}{\rho \vdash e_1 \,@\, e_2 : t\ \texttt{list}}$$

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

Angiv et typetræ for udtrykket

```
let x = [43] in x @ [3+4] end
```

Du finder to eksempler på typetræer i figur 4.8 og 4.9 på side 72 i PLC.