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

9. januar 2023

Indhold

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

0 Foregående kommentarer

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

```
diff filnavn.filtype
@@ -1,2 +1,2 @@ evt. toppen af nuværende kodeblok
 Gammel, ændret linje
-Fjernet eller modificeret linje før ændring
+Ny eller modificeret linje efter ændring
```

Den første linje visen filen, der er blevet ændret. Derefter vises linjetallet på ændringen. Dernæst er selve ændringen.

Derudover følger afsnittenes numre opgaverne i eksamenssættet. Altså svarer afsnit 2.4 til opgave 2, underopgave 4.

1 Icon

1.1 Tallene 5-12

Forneden ses den abstrakte syntaks i Icon til at udskrive samtlige tal fra 5-12:

```
> open Icon;;
> let numbers = FromTo(5, 12);;
val numbers : expr = FromTo (5, 12)

> run (Every (Write numbers));;
5 6 7 8 9 10 11 12 val it : value = Int 0
```

Generatoren `numbers` producerer alle tal fra 5-12. Operationen `Write` udskriver 1 værdi fra generatoren til skærmen. Her bruges `Every` til at køre *alle* elementerne i `number` gennem `Write`.

1.2 Tallene 5-12 større end 10

Forneden ses den abstrakte syntaks i Icon til at udskrive tallene fra 5-12, som er større end 10:

```
> run (Every (Write (Prim("<", CstI 10, numbers))));;
11 12 val it : value = Int 0
```

Koden virker, fordi den primitive operation $a < b$ filtrerer elementer fra b fra, hvis ikke $a < b_i$. Altså bliver alle elementer fra `numbers`, hvor 10 ikke er strengt mindre end elementet, filtreret fra. Resten er det samme som det foregående Icon-udtryk.

1.3 Intervallerne 6-12, 7-12, ..., 12

Forneden ses den abstrakte syntaks i Icon til at udskrive intervallerne 6-12, 7-12, 8-12, ..., 12:

```
> run (Every (Write (Prim("<", numbers, And(Write(CstS "\n"), numbers)))));;

 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
```

Der er foretaget 2 ændringer fra det foregående Icon-udtryk:

Frem for at første argument til primitivet < er en 1-elements generator `CstI 10`, er det i

stedet en generator med intervallet 5-12 (`numbers`). Det vil sige, at for hver element i 5-12, vil generatoren til højre (også `numbers`) blive kørt igennem primitivet <. For hver gennemkørsel bliver samme filtreringsproces som før gennemgået — nu med 5, 6, 7, ..., 12 i stedet for kun 10.

Derudover er det højre `numbers` ændret til `And(Write(CstS "\n"), numbers)`. And-reglen producerer alle elementer fra højre generator for hvert element i venstre generator. Fordi venstre generator kun har 1 element, producerer sammensætningen præcis de samme elementer, som `numbers` normalt ville. Men forskellen er, at `Write(CstS "\n")` bliver kørt, hvilket skriver en ny linje til skærmen, hver gang `numbers` køres igennem.

1.4 Tegnintervaller

Kodeændringerne i `Icon.fs` for at understøtte tegnintervaller er indsat herunder. En forklaring følger.

```
1  diff Icon.fs
2  @@ -28,6 +28,7 @@ type expr =
3     | CstI of int
4     | CstS of string
5     | FromTo of int * int
6  +  | FromToChar of char * char
7     | Write of expr
8     | If of expr * expr * expr
9     | Prim of string * expr * expr
10 @@ -67,6 +68,16 @@ let rec eval (e : expr) (cont : cont) (econt : econt) =
11            else
12                econt ()
13        loop i1
14 +    | FromToChar(c1, c2) ->
15 +      let nextChar = int >> (+) 1 >> char;
16 +
17 +      let rec nextChars cur =
18 +          if cur <= c2 then
19 +              cont (Str (string cur)) (fun () -> nextChar cur |> nextChars)
20 +          else
21 +              econt ()
22 +
23 +      nextChars c1
24      | Write e ->
25        eval e (fun v -> fun econt1 -> (write v; cont v econt1)) econt
26      | If(e1, e2, e3) ->
```

Typen `expr` er udvidet med varianten `FromToChar`, som holder på et par af tegn, ligesom `FromTo` holder på et par af heltal. Linje 15 viser en hjælpefunktion, som finder det næste tegn i sekvensen ved blot at plusse tegnets numeriske værdi med 1. Herefter bruges en rekursiv closure til at iterere gennem hele sekvensen, så længe det nuværende element er mindre end eller lig med det sidste element. Hvert rekursive kald sker i en anonym closure, der gives som

argument til den givne continutation — derfor kommer evalueringen til at være lazy. Lige så snart `if`-sætningen ikke er sand, fejler generatoren.

Eksemplerne fra opgaven, samt et ekstra eksempel med intervallet `!-?`, er indsat således:

```
> let chars = FromToChar('C', 'L');;
val chars : expr = FromToChar ('C', 'L')

> 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

> run (Every(Write(FromToChar('!', '?'))));;
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? val it : value = Int 0
```

1.5 Sammenligning af strenge

Ændringen i `Icon.fs` er som følgende:

```
diff Icon.fs
@@ -96,6 +96,11 @@ let rec eval (e : expr) (cont : cont) (econt : econt) =
                         cont (Int i2) econt2
                     else
                         econt2 ()
+             | ("<", Str s1, Str s2) ->
+                 if s1 < s2 then
+                     cont (Str s2) econt2
+                 else
+                     econt2 ()
              | _ -> Str "unknown prim2")
              econt1)
          econt
```

Logikken er den samme som for heltal — kun typerne er anderledes. Ved at matche på `Str`-varianten, bliver den nye kode kun kørt, når begge argumenter er strenge. Hvis første argument er strengt mindre end andet argument, bliver andet argument outputtet af generatoren. Ellers fejler generatoren. Det demonstreres, at eksemplerne fungerer:

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

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

1.6 Intervallet H-L

Icon-udtrykket til at skrive H-L til skærmen ved at bruge `chars` kan ses her:

```
> run (Every(Write(Prim("<", CstS "G", chars))));;
H I J K L val it : value = Int 0
```

Løsningen fungerer ved at tage intervallet `chars` (C til L) og filtrere alle strenge fra, hvor G ikke er strengt mindre. Betingelsen bliver først sand fra H og op. `chars` slutter på L, så den resulterende generator kommer til at være intervallet H-L.

2 List-C: Stack

2.1 Lexer og parser

Ændringerne til lexeren og parseren er som følgende:

```
1  diff CLex.fsl
2  @@ -37,2 +37,6 @@ let keyword s =
3       | "while"       -> WHILE
4  +    | "createStack" -> CREATESTACK
5  +    | "pushStack"   -> PUSHSTACK
6  +    | "popStack"    -> POPSTACK
7  +    | "printStack"  -> PRINTSTACK
8       | _             -> NAME s
9
10 diff CPar.fsy
11 @@ -17,2 +17,3 @@ let nl = CstI 10
12  %token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE
13 +%token CREATESTACK PUSHSTACK POPSTACK PRINTSTACK
14  %token NIL CONS CAR CDR DYNAMIC SETCAR SETCDR
15 @@ -140,2 +141,3 @@ AtExprNotAccess:
16       Const                              { CstI $1               }
17 +   | StackOperations                    { $1                    }
18     | LPAR ExprNotAccess RPAR            { $2                    }
19 @@ -150,2 +152,9 @@ AtExprNotAccess:
20
21 +StackOperations:
22 +      CREATESTACK LPAR Expr RPAR          { Prim1("createStack", $3)   }
23 +    | PUSHSTACK LPAR Expr COMMA Expr RPAR { Prim2("pushStack", $3, $5) }
24 +    | POPSTACK LPAR Expr RPAR             { Prim1("popStack", $3)      }
25 +    | PRINTSTACK LPAR Expr RPAR           { Prim1("printStack", $3)    }
26 +;
27
28  Access:
```

4 tokens er tilføjet på linje 13. Linje 4-7 viser lexerændringerne, der oversætter keywords til tokens. Linje 17 viser en ny grammatikregel `StackOperations`, som defineres på linje 21-26. Det er ikke nødvendigt at dedikere en separat blok til `StackOperations` — man kunne sagtens have indsat de 4 nye grammatikregler under `AtExprNotAccess`, men jeg flyttede det ud, for at gøre koden mere overskuelig.

Derefter har jeg ændret `ListCC.fs`, så det abstrakte syntakstræ bliver skrevet til skærmen:

```
1  diff ListCC.fs
2  @@ -13,3 +13,6 @@ let _ = if args.Length > 1 then
3              printf "Compiling %s to %s\n" source target;
4  -           try ignore (Comp.compileToFile (Parse.fromFile source) target)
5  +           try
```

```
+                let s = Parse.fromFile source
+                printfn "Parsed: %A" s
+                ignore (Comp.compileToFile (s) target)
             with Failure msg -> printf "ERROR: %s\n" msg
```

Efter `listcc.exe` er bygget med de nye ændringer, parses `stack.lc` således:

```
$ mono listcc.exe stack.lc
List-C compiler v 1.0.0.0 of 2012-02-13
Compiling stack.lc to stack.out
Parsed: 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 "s")))));
         Stmt (Expr (Prim1 ("printStack", Access (AccVar "s"))))])]
ERROR: unknown primitive 1
```

Bemærk at fejlen `ERROR: unknown primitive 1` er forventet, idet compileren ikke er udvidet med logikken til at håndtere stack-primitiver endnu.

Derudover differentierer linje 14 sig fra opgavens "forventede" uddata, idet det givne eksempel på uddata har en variabel med navnet `t` i stedet for `s`. Jeg antager, at dette er en fejl i opgaveformuleringen.

2.2 Implementering af stak i bytekodefortolkeren

Kodeændringerne beskrives først (i rækkefølgen, ændringerne vises), hvorefter de vises:

- Linje 5: En lille ændring, så bytekodefortolkeren virker på macOS med ARM-processorer.
- Linje 9: Et tag, der bruges i headeren, når et objekt allokeres på hoben. Det bruges til at vide, at typen på det allokerede objekt er en stak.
- Linje 13-16: Bytekodeinstruktionernes tal forbindes med deres navn.
- Linje 20-23: Kode til at udskrive bytekodeinstruktionerne.
- Linje 27-32: En hjælperfunktion til at udskrive værdier afhængigt af deres tag (heltal og referencer).

- Linje 37-53: Implementering af `createStack`. Der sørges for at N fra stakken bliver untagget — og hvis N er negativt, fejler programmet. Et objekt med størrelsen $N+2$ bliver allokeret, da der skal være plads til N elementer samt 2 pladser til N og `top`. Headeren bliver tilføjet af `allocate`. Derefter initialiseres stakken med N, `top` som 0, og resten af pladserne udfyldes med et 0 (tagget som heltal). Adressen tilføjes til programstakken.
- Linje 54-68: Implementering af `pushStack`. Værdien, der skal skubbes på stakken, læses (uden at untagge den, så typen bliver bibeholdt), og selve stakkens adresse læses også. Derfra læses N og `top` fra hoben. Hvis stakken er fuld, fejler programmet. Stakkens adresse efterlades på programstakken.
- Linje 69-80: Implementering af `popStack`. Stakkens adresse læses fra programstakken. Derfra læses `top` fra hoben. Hvis stakken er tom, fejler programmet. Værdien af `top` dekrementeres med 1, og den poppede værdi efterlades på programstakken.
- Linje 81-90: Implementering af `printStack`. Stakkens adresse læses fra programstakken (uden at poppe den). Derfra læses og udskrives N og `top` fra hoben. Ved brug af en løkke og hjælpefunktionen `printWord` udskrives alle elementer i stakken typeafhængigt.
- `Machine.fs` er blot udvidet med de nye instruktioner og bytekode-tal.

```
diff listmachine.c
@@ -83,3 +83,3 @@ created when allocating all but the last word of a free block.

-#if _WIN64 || __x86_64__ || __ppc64__
+#if _WIN64 || __x86_64__ || __ppc64__ || __aarch64__
   #define ENV64
@@ -153,2 +153,3 @@ word* readfile(char* filename);
 #define CONSTAG 0
+#define STACKTAG 1

@@ -197,2 +198,6 @@ word *freelist;
 #define SETCDR 31
+#define CREATESTACK 32
+#define PUSHSTACK   33
+#define POPSTACK    34
+#define PRINTSTACK  35

@@ -239,2 +244,6 @@ void printInstruction(word p[], word pc) {
   case SETCDR: printf("SETCDR"); break;
+  case CREATESTACK: printf("CREATESTACK"); break;
+  case PUSHSTACK: printf("PUSHSTACK"); break;
+  case POPSTACK: printf("POPSTACK"); break;
+  case PRINTSTACK: printf("PRINTSTACK"); break;
   default:     printf("<unknown>"); break;
@@ -243,2 +252,9 @@ void printInstruction(word p[], word pc) {

```

```
+void printWord(word w) {
+  if (IsInt(w))
+    printf(WORD_FMT " ", Untag(w));
+  else
+    printf("#" WORD_FMT " ", w);
+}
+
 // Print current stack (marking heap references by #) and current instruction
@@ -381,2 +397,56 @@ int execcode(word p[], word s[], word iargs[], int iargc,
  ↪  int /* boolean */ trac
     } break;
+    case CREATESTACK: {
+      word N = Untag(s[sp--]);
+
+      if (N < 0) {
+        printf("Cannot create stack with negative size\n");
+        return -1;
+      }
+
+      word* stack = allocate(STACKTAG, N + 2, s, sp);
+
+      stack[1] = N;
+      stack[2] = 0;
+      for (int i = 3; i <= N + 2; i++)
+        stack[i] = Tag(0);
+
+      s[++sp] = (word) stack;
+    } break;
+    case PUSHSTACK: {
+      word v = s[sp--];
+      word* stack = (word*) s[sp];
+
+      word N = stack[1];
+      word top = stack[2];
+
+      if (top >= N) {
+        printf("Stack is full - cannot push more items\n");
+        return -1;
+      }
+
+      stack[2] = top + 1;
+      stack[top + 3] = v;
+    } break;
+    case POPSTACK: {
+      word* stack = (word*) s[sp--];
+      word top = stack[2];
+
+      if (top <= 0) {
+        printf("Stack is empty - cannot pop\n");
```

```
+        return -1;
+      }
+
+      stack[2] = top - 1;
+      s[++sp] = stack[top + 3 - 1];
+    } break;
+    case PRINTSTACK: {
+      word* stack = (word*) s[sp];
+      word N = stack[1];
+      word top = stack[2];
+
+      printf("STACK(" WORD_FMT ", " WORD_FMT ")[ ", N, top);
+      for (int i = 3; i < top + 3; i++)
+        printWord(stack[i]);
+      printf("]\n");
+    } break;
     default:

diff Machine.fs
@@ -49,2 +49,6 @@ type instr =
   | SETCDR                              (* set second field of cons cell   *)
+  | CREATESTACK
+  | PUSHSTACK
+  | POPSTACK
+  | PRINTSTACK

@@ -105,2 +109,6 @@ let CODESETCAR = 30;
 let CODESETCDR = 31;
+let CODECREATESTACK = 32;
+let CODEPUSHSTACK   = 33;
+let CODEPOPSTACK    = 34;
+let CODEPRINTSTACK  = 35;

@@ -145,2 +153,6 @@ let makelabenv (addr, labenv) instr =
     | SETCDR         -> (addr+1, labenv)
+    | CREATESTACK    -> (addr+1, labenv)
+    | PUSHSTACK      -> (addr+1, labenv)
+    | POPSTACK       -> (addr+1, labenv)
+    | PRINTSTACK     -> (addr+1, labenv)

@@ -183,2 +195,6 @@ let rec emitints getlab instr ints =
     | SETCDR         -> CODESETCDR :: ints
+    | CREATESTACK    -> CODECREATESTACK :: ints
+    | PUSHSTACK      -> CODEPUSHSTACK   :: ints
+    | POPSTACK       -> CODEPOPSTACK    :: ints
+    | PRINTSTACK     -> CODEPRINTSTACK  :: ints
```

2.3 Implementering af stak i oversætteren

Kodeændringen er vist forneden. 1-arguments-primitivet er udvidet med `createStack`, `popStack`, og `printStack`. 2-arguments-primitivet er udvidet med `pushStack`. De foregående værdier på stakken, som de nye instruktioner har brug for, tilføjes i forvejen af `cExpr`.

```
diff Comp.fs
@@ -182,2 +182,5 @@ and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) :
 ↪  instr list =
           | "cdr"    -> [CDR]
+          | "createStack" -> [CREATESTACK]
+          | "popStack"    -> [POPSTACK]
+          | "printStack"  -> [PRINTSTACK]
           | _        -> raise (Failure "unknown primitive 1"))
@@ -201,2 +204,3 @@ and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) :
 ↪  instr list =
           | "setcdr" -> [SETCDR]
+          | "pushStack" -> [PUSHSTACK]
           | _        -> raise (Failure "unknown primitive 2"))
```

Ved at køre bytekodefortolkeren op mod `stack.lc` (hvor `stack.out` kommer fra de nye ændringer i oversætteren) fås:

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

Used 0 cpu milli-seconds
```

Opgavebeskrivelsen siger, at stakken skal udskrives som `STACK(3, 2)[ 42 43 ]`, men det givne eksempel viser `STACK(3, 2)=[ 42 43 ]` (bemærk det tilføjede lighedstegn). Jeg har antaget, at jeg skal følge opgavebeskrivelsen, og jeg har derfor ikke printet et lighedstegn.

2.4 Test af stakken

De vigtigste ting at teste er, at stakken fungerer som forventet i de mest sædvanlige tilfælde, samt at den fungerer i edge cases. Fordi `stack.lc` ikke tester edge cases, vil jeg fokusere på dem.

1. `createStack` bør virke, når størrelsen er 0:

```
void main() {
  printStack(createStack(0));
}
```

   Uddata bekræfter, at testen består:

```
$ ./listmachine test-createStack-empty.out
STACK(0, 0)[ ]

Used 0 cpu milli-seconds
```

2. `createStack` bør fejle, når størrelsen er negativ:

```
void main() {
  printStack(createStack(-1));
}
```

Uddata viser, at programmet fejler (som forventet):

```
$ ./listmachine test-createStack-negative.out
Cannot create stack with negative size

Used 0 cpu milli-seconds
```

3. `pushStack` bør bevare typen på den givne værdi — så heltal opbevares som heltal, og referencer til hoben opbevares som referencer:

```
void main() {
  dynamic s;
  s = createStack(3);

  pushStack(s, s);
  pushStack(s, 100);

  printStack(s);

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

Uddata viser, at bytekodefortolkeren differentierer mellem referencer til hoben og skalarværdier internt, samt at værdierne korrekt poppes ud:

```
$ ./listmachine test-pushStack-types.out
STACK(3, 2)[ #5813341184 100 ]
100 5813341184
Used 0 cpu milli-seconds
```

4. `pushStack` bør fejle, når der pushes til en fuld stak:

```
void main() {
  dynamic s;
  s = createStack(1);
  pushStack(s, 1);
  printStack(s);
  pushStack(s, 2);
  printStack(s);
}
```

Uddata viser, at programmet fejler (som forventet):

```
$ ./listmachine test-pushStack-pushfull.out
STACK(1, 1)[ 1 ]
Stack is full - cannot push more items

Used 0 cpu milli-seconds
```

3 Micro-C: Tupler

3.1 Abstrakt syntaks

Kodeændringerne ser ud som følgende:

```
diff Absyn.fs
@@ -14,2 +14,3 @@ type typ =
    | TypP of typ                       (* Pointer type                *)
+   | TypT of typ * int option

@@ -30,2 +31,3 @@ and access =
    | AccIndex of access * expr         (* Array indexing         a[e] *)
+   | TupIndex of access * expr
```

Det bekræftes, at den abstrakte syntaks kan konstrueres i F#-fortolkeren:

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

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

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

3.2 Lexer

Kodeændringerne ser ud som følgende for lexeren:

```
diff CLex.fsl
@@ -74,2 +74,4 @@ rule Token = parse
    | ']'              { RBRACK }
+   | "(|"             { LBARPAR }
+   | "|)"             { RBARPAR }
    | ';'              { SEMI }
```

Bemærk, at koden ikke kommer til at virke, før parseren er opdateret.

3.3 Parserspecifikation

Til første del er kodeændringen:

```
diff CPar.fsy
@@ -21,2 +21,3 @@ let nl = CstI 10
```

```
 %token LPAR RPAR LBRACE RBRACE LBRACK RBRACK SEMI COMMA ASSIGN AMP
+%token LBARPAR RBARPAR
 %token EOF
@@ -32,2 +33,3 @@ let nl = CstI 10
 %nonassoc NOT AMP
+%nonassoc LBARPAR
 %nonassoc LBRACK          /* highest precedence  */
@@ -63,2 +65,4 @@ Vardesc:
    | Vardesc LBRACK CSTINT RBRACK    { compose1 (fun t -> TypA(t, Some $3)) $1 }
+   | Vardesc LBARPAR RBARPAR         { compose1 (fun t -> TypT(t, None)) $1    }
+   | Vardesc LBARPAR CSTINT RBARPAR  { compose1 (fun t -> TypT(t, Some $3)) $1 }
 ;
```

Følgende kommandolinje-udsnit viser, at lexer og parser kan bygges uden fejl, samt at eksemplerne parses som forventet.

```
$ fslex --unicode CLex.fsl
compiling to dfas (can take a while...)
64 states
writing output

$ fsyacc --module CPar CPar.fsy
        building tables
computing first function...        time: 00:00:00.0633739
building kernels...        time: 00:00:00.0500669
building kernel table...        time: 00:00:00.0099092
computing lookahead relations.....................        time: 00:00:00.0534733
building lookahead table...        time: 00:00:00.0179177
building action table...        time: 00:00:00.0275403
        building goto table...        time: 00:00:00.0052777
        returning tables.
        142 states
        22 nonterminals
        44 terminals
        74 productions
        #rows in action table: 142

$ fsharpi ...
...
> 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")])]
```

Til anden del er kodeændringen yderligere:

```
diff CPar.fsy
@@ -153,2 +153,3 @@ Access:
   | Access LBRACK Expr RBRACK           { AccIndex($1, $3)    }
+  | Access LBARPAR Expr RBARPAR         { TupIndex($1, $3)    }
```

Igen vises det, at lexer og parser kan bygges og køres uden fejl:

```
$ fslex --unicode CLex.fsl
compiling to dfas (can take a while...)
64 states
writing output

$ fsyacc --module CPar CPar.fsy
        building tables
computing first function...        time: 00:00:00.0670778
building kernels...        time: 00:00:00.0549343
building kernel table...        time: 00:00:00.0115684
computing lookahead relations.....................        time: 00:00:00.0597363
building lookahead table...        time: 00:00:00.0192274
building action table...        time: 00:00:00.0300694
        building goto table...        time: 00:00:00.0059809
        returning tables.
        145 states
        22 nonterminals
        44 terminals
        75 productions
        #rows in action table: 145

$ fsharpi ...
...
> open ParseAndComp;;
> 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)))))])]
```

3.4 Udvidelse af oversætteren

Først kommer en forklaring af kodeændringerne, og derefter vises koden.

Mine første ændringer er at tilføje flere match-betingelser, så man ikke kan allokere arrays af tupler, tupler af tupler, eller tupler af arrays. Således sikres det, at de opbevarede elementer hvert kun fylder 1 ord.

Derefter implementeres allokering af tupler. Koden minder om koden til at allokere variable, men i stedet for at allokere 1 ord, allokeres i ord, så der er plads til alle elementer. Dette er forskelligt fra arrays, der allokerer $i+1$ ord, så der også er plads til en peger. Allokeringen sker ved, at variabelmiljøet opdateres, så det nye offset er i højere. Derefter inkrementeres stakpegeren med i, så tuplens stakpladser ikke bliver overskrevet af efterfølgende instruktioner.

`cAccess` er udvidet til at håndtere tupel-indeksering. Forskellen fra arrays er, at man ikke har en `LDI`-instruktion, efter stak-adressen for `acc` er beregnet. Grunden er, at `acc` for arrays svarer til en peger til første element — men for tupler er `acc` allerede adressen på første element. Derfor er løsningen blot at fjerne `LDI` og plusse tuplens adresse med indekset.

```
diff Comp.fs
@@ -71,4 +71,6 @@ let allocate (kind : int -> var) (typ, x) (varEnv : varEnv) :
 ↪  varEnv * instr lis
     | TypA (TypA _, _) ->
       raise (Failure "allocate: array of arrays not permitted")
+    | TypA (TypT _, _) ->
+      raise (Failure "allocate: array of tuples not permitted")
     | TypA (t, Some i) ->
@@ -76,4 +78,12 @@ let allocate (kind : int -> var) (typ, x) (varEnv : varEnv) :
 ↪  varEnv * instr lis
       let code = [INCSP i; GETSP; CSTI (i-1); SUB]
       (newEnv, code)
+    | TypT (TypA _, _) ->
+      raise (Failure "allocate: tuple of arrays not permitted")
+    | TypT (TypT _, _) ->
+      raise (Failure "allocate: tuple of tuples not permitted")
+    | TypT (t, Some i) ->
+      let newEnv = ((x, (kind fdepth, typ)) :: env, fdepth + i)
+      let code = [INCSP i]
+      (newEnv, code)
     | _ ->
       let newEnv = ((x, (kind (fdepth), typ)) :: env, fdepth+1)
@@ -220,4 +230,7 @@ and cAccess access varEnv funEnv : instr list =
     | AccIndex(acc, idx) -> cAccess acc varEnv funEnv
                             @ [LDI] @ cExpr idx varEnv funEnv @ [ADD]
+    | TupIndex(acc, idx) -> cAccess acc varEnv funEnv
+                            @ cExpr idx varEnv funEnv
+                            @ [ADD]
```

3.5 Udklip og modifikation af kode

Jeg har forstået opgavebeskrivelsen som om, at man undervejs gerne må vise udklip af koden og forklare dem. Derfor har jeg gjort det, frem for at indsætte det hele under dette afsnit. Se tidligere afsnit.

3.6 Kørsel af programmet

Det dokumenteres, at programmet producerer det korrekte resultat:

```
$ java Machine.java tuple.out
55 56 55 56
Ran 0.001 seconds
```

For komplethedens skyld demonstreres det også, at resultatet bliver gemt på stakken som forventet. `Machinetrace` viser, at præcis 2 pladser på stakken bruges på at opbevare tuplen (linje 4) — altså allokeres der ikke for meget eller for lidt plads. De sidste par linjer viser, at værdien 55 bliver skrevet til første plads, samt at tuplens to pladser ikke bliver overskrevet, på nær ved `STI`-operationer.

```
$ java Machinetrace tuple.out
[ ]{0: LDARGS}
[ ]{1: CALL 0 5}
[ 4 -999 ]{5: INCSP 2}
[ 4 -999 0 0 ]{7: GETBP}
[ 4 -999 0 0 2 ]{8: CSTI 0}
[ 4 -999 0 0 2 0 ]{10: ADD}
[ 4 -999 0 0 2 ]{11: CSTI 0}
[ 4 -999 0 0 2 0 ]{13: ADD}
[ 4 -999 0 0 2 ]{14: CSTI 55}
[ 4 -999 0 0 2 55 ]{16: STI}
[ 4 -999 55 0 55 ]{17: INCSP -1}
[ 4 -999 55 0 ]{19: GETBP}
...
```

4 Micro-ML: Lists

4.1 Lexer og parser

Ændringerne er som følgende:

1. `List`-typen er tilføjet til den abstrakte syntaks.
2. De nye tokens er tilføjet til lexeren og parseren.
3. `@` er givet samme associativitet og præcedens som `+`.
4. Der er tilføjet en simpel grammatikregel, der genkender `Expr @ Expr`.
5. Der er tilføjet en regel til at genkende lister. Den starter med `LBRACK ListItems RBRACK`, hvor `ListItems` er defineret længere nede. Reglen genkender ikke-tomme lister, fordi den altid som minimum har én `Expr`. Hvis den har flere, bruger reglen rekursivt sig selv til at parse kommaseparerede udtryk. Fordi "kaldet" til den rekursive regel er på højre side, bliver listen parset venstre-til-højre, og elementerne kommer derfor i den rigtige rækkefølge.

I kode er ændringerne:

```
diff Absyn.fs
@@ -6,4 +6,5 @@ type expr =
    | CstI of int
    | CstB of bool
+   | List of expr list
    | Var of string
    | Let of string * expr * expr

diff FunLex.fsl
@@ -56,4 +56,8 @@ rule Token = parse
    | '('              { LPAR }
    | ')'              { RPAR }
+   | '['              { LBRACK }
+   | ']'              { RBRACK }
+   | '@'              { AT }
+   | ','              { COMMA }
    | eof              { EOF }
    | _                { failwith "Lexer error: illegal symbol" }

diff FunPar.fsy
@@ -15,5 +15,6 @@
 %token PLUS MINUS TIMES DIV MOD
 %token EQ NE GT LT GE LE
 %token LPAR RPAR
+%token LBRACK RBRACK COMMA AT
 %token EOF

```

```
@@ -21,5 +22,5 @@
 %left EQ NE
 %left GT LT GE LE
-%left PLUS MINUS
+%left PLUS MINUS AT
 %left TIMES DIV MOD
 %nonassoc NOT           /* highest precedence  */
@@ -28,4 +29,5 @@
 %type <Absyn.expr> Main Expr AtExpr Const
 %type <Absyn.expr> AppExpr
+%type <Absyn.expr list> ListItems

 %%
@@ -51,4 +53,5 @@ Expr:
   | Expr GE    Expr                    { Prim(">=", $1, $3)     }
   | Expr LE    Expr                    { Prim("<=", $1, $3)     }
+  | Expr AT    Expr                    { Prim("@",  $1, $3)     }
 ;

@@ -58,7 +61,13 @@ AtExpr:
   | LET NAME EQ Expr IN Expr END        { Let($2, $4, $6)        }
   | LET NAME NAME EQ Expr IN Expr END   { Letfun($2, $3, $5, $7) }
+  | LBRACK ListItems RBRACK             { List($2)               }
   | LPAR Expr RPAR                      { $2                     }
 ;

+ListItems:
+    Expr COMMA ListItems { $1 :: $3 }
+  | Expr                 { [$1]     }
+;
+
 AppExpr:
     AtExpr AtExpr                       { Call($1, $2)           }
```

Alle eksemplerne fra opgaven (`ex01`-`ex06`) er testet, og de giver alle det forventede resultat:

ex01

```
> 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])))
```

ex02

```
> fromString "let l = [] in l end";;
System.Exception: parse error near line 1, column 10
  at Microsoft.FSharp.Core.PrintfModule+PrintFormatToStringThenFail@1433[T┘
 ↪  Result].Invoke (System.String message) [0x00000] in
 ↪  <b56f33d2f53c2e7533e6754e4d8591b5>:0
  at FSI_0002.Parse.fromString (System.String str) [0x0009a] in
 ↪  <c1a46470212049aa91eab2ba84025ea4>:
```

ex03

```
> fromString "let l = [43] in l @ [3+4] end";;
val it : Absyn.expr =
  Let
    ("l", List [CstI 43],
     Prim ("@", Var "l", List [Prim ("+", CstI 3, CstI 4)]))
```

ex04

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

ex05

```
> fromString "let f x = x+1 in [f] end";;
val it : Absyn.expr =
  Letfun ("f", "x", Prim ("+", Var "x", CstI 1), List [Var "f"])
```

ex06

```
> fromString "let id x = x in [id] end";;
val it : Absyn.expr = Letfun ("id", "x", Var "x", List [Var "id"])
```

4.2 Implementering af fortolker

Ændringerne er som følgende:

1. Typen `ListV` er tilføjet. Internt holder den på en F#-liste af værdier.
2. Når fortolkeren støder på en `List`, kører den alle listens udtryk igennem, evaluerer dem, og danner en `ListV` med resultaterne jf. evalueringsreglen *list*.
3. Ved den primitive operation `@`, matches der på 2 liste-argumenter. Her sikrer typerne i matchet, at kun 2 lister kan sammensættes. Internt bruges F#'s egen `@`, der selv opfylder evalueringsreglen @.
4. Ved den primitive operation =, matches der på 2 liste-argumenter. Her sikrer typerne i matchet, at 2 lister kan sammenlignes (man kan for eksempel ikke sammenligne en liste med et tal ifølge evalueringsreglerne). Igen bruges F#'s egen sammenligningsoperator =, som opfylder regel $=T$ og $=F$.

Koden er vist nedenfor.

```
diff HigherFun.fs
@@ -31,4 +31,5 @@ type value =
   | Int of int
   | Closure of string * string * expr * value env       (* (f, x, fBody,
 ↪  fDeclEnv) *)
+  | ListV of value list

 let rec eval (e : expr) (env : value env) : value =
@@ -36,4 +37,5 @@ let rec eval (e : expr) (env : value env) : value =
     | CstI i -> Int i
     | CstB b -> Int (if b then 1 else 0)
+    | List l -> l |> List.map (fun e' -> eval e' env) |> ListV
     | Var x  -> lookup env x
     | Prim(ope, e1, e2) ->
@@ -46,4 +48,6 @@ let rec eval (e : expr) (env : value env) : value =
       | ("=", Int i1, Int i2) -> Int (if i1 = i2 then 1 else 0)
       | ("<", Int i1, Int i2) -> Int (if i1 < i2 then 1 else 0)
+      | ("@", ListV l1, ListV l2) -> ListV (l1 @ l2)
+      | ("=", ListV l1, ListV l2) -> Int (if l1 = l2 then 1 else 0)
       |  _ -> failwith "unknown primitive or wrong type"
     | Let(x, eRhs, letBody) ->
```

Alle tests fra opgaven er testet igen med `run`. Til hvert test er tilknyttet en kommentar, der forklarer, hvorfor resultatet er korrekt.

ex01 Sammensætningen af `[2, 3]` og `[1, 4]` bliver til listen `[2, 3, 1, 4]`. Programmet evalueres til 1 (sand), hvilket er forventet.

```
> fromString @"let l1 = [2, 3] in
-   let l2 = [1, 4] in
-     l1 @ l2 = [2, 3, 1, 4]
-   end
- end" |> run;;
val it : HigherFun.value = Int 1
```

ex02 Parseren fejler som forventet stadig.

```
> fromString "let l = [] in l end" |> run;;
System.Exception: parse error near line 1, column 10
  at Microsoft.FSharp.Core.PrintfModule+PrintFormatToStringThenFail@1433[T┘
 ↪  Result].Invoke (System.String message) [0x00000] in
 ↪  <b56f33d2f53c2e7533e6754e4d8591b5>:0
  at FSI_0002.Parse.fromString (System.String str) [0x0009a] in
 ↪  <2700d72b466e4c8e82c56e4c7d92bdac>:0
```

ex03 Sammensætningen af `[43]` og `[3+4]` bør blive til listen `[43, 7]`, hvilket programmet outputter. Dette eksempel bekræftes også.

```
> fromString "let l = [43] in l @ [3+4] end" |> run;;
val it : HigherFun.value = ListV [Int 43; Int 7]
```

ex04 Sammensætningen af `[3]` og `[3]` bliver til listen `[3, 3]`, hvilket ikke er det samme som `[7]`. Programmet outputter derfor korrekt 0 (falsk).

```
> fromString "let l = [3] in l @ [3] = [3+4] end" |> run;;
val it : HigherFun.value = Int 0
```

ex05 `f` er en closure, der ikke er kaldt endnu, når den indsættes i listen. Det er derfor korrekt, at resultatet er en liste med en closure som element.

```
> fromString "let f x = x+1 in [f] end" |> run;;
val it : HigherFun.value =
  ListV [Closure ("f", "x", Prim ("+", Var "x", CstI 1), [])]
```

ex06 Samme som ovenstående.

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

4.3 Typetræ

Nedenfor er et typetræ til udtrykket `let x = [43] in x @ [3+4] end` udformet. Reglerne med præfiks *p* henviser til typereglerne på Figur 6.1 i Peter Sestofts *Programming Language Concepts*.

Bemærk, at jeg har brugt syntaksen $\rho' = \rho[\texttt{x} \mapsto \texttt{int list}] \vdash \ldots$ til at definere ρ' som alias for $\rho[\texttt{x} \mapsto \texttt{int list}]$. Det er gjort for at få træet til at fylde mindre på siden.

$$\dfrac{\dfrac{\dfrac{}{\rho \vdash \texttt{43} : \texttt{int}}\,(p1)}{\rho \vdash \texttt{[43]} : \texttt{int list}}\,(list) \qquad \dfrac{\dfrac{\rho'(\texttt{x}) = \texttt{int list}}{\rho' \vdash \texttt{x} : \texttt{int list}}\,(p3) \qquad \dfrac{\dfrac{\dfrac{}{\rho' \vdash \texttt{3} : \texttt{int}}\,(p1) \quad \dfrac{}{\rho' \vdash \texttt{4} : \texttt{int}}\,(p1)}{\rho' \vdash \texttt{3+4} : \texttt{int}}\,(p4)}{\rho' \vdash \texttt{[3+4]} : \texttt{int list}}\,(list)}{\rho' = \rho[\texttt{x} \mapsto \texttt{int list}] \vdash \texttt{x @ [3+4]} : \texttt{int list}}\,(@)}{\rho \vdash \texttt{let x = [43] in x @ [3+4] end} : \texttt{int list}}\,(p6)$$