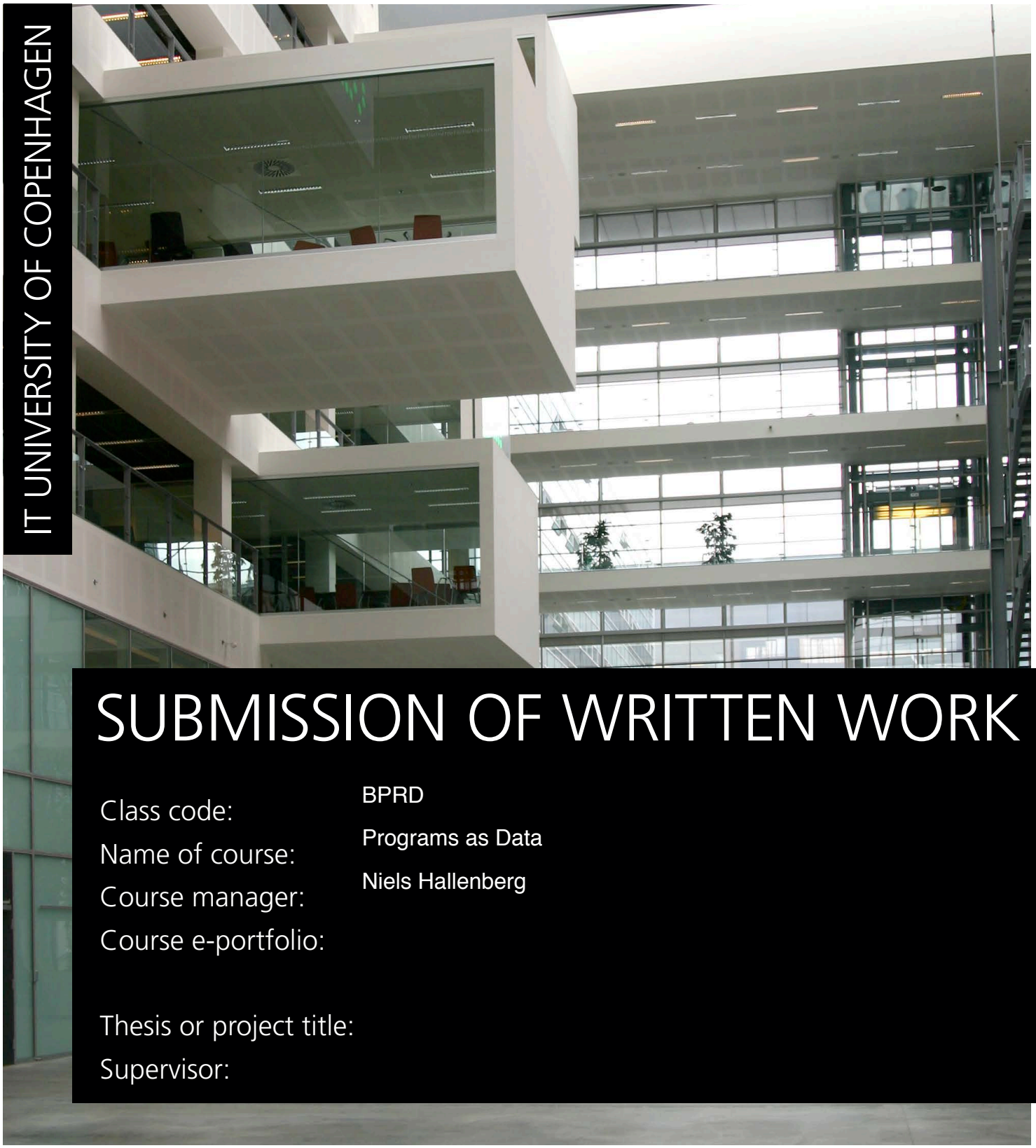
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code: BPRD

Name of course: Programs as Data

Course manager: Niels Hallenberg

Course e-portfolio:

Thesis or project title:

Supervisor:

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Jacob Benjamin Cholewa	29/03-1992	jbec@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Programmer som Data

Eksamens opgave Januar 2015

Jacob Benjamin Cholewa
jbec@itu.dk

January 12, 2015

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

Jacob Cholewa

Signature

12/01 2015

Date

Jacob Benjamin Cholewa jbec@itu.dk

Opgave 1

Spørgsmål 1.1

Det regulære udtryk $e(fd*) * x$ beskriver et bar system. Du kan gå ind i baren, $e(nter)$, og her kan du så hente øl, $f(etch)$, (Du behøver ikke). Når du har en øl kan du hente en mere eller du kan drikke af den du allerede har 0 til flere gange $d(rink)$. Tilsidst kan du gå ud af baren, $e(xit)$, lige gyldigt om du har eller ikke har hentet og drukket øl.

Dette regulære udtryk beskriver altså for eksempel disse strenge:

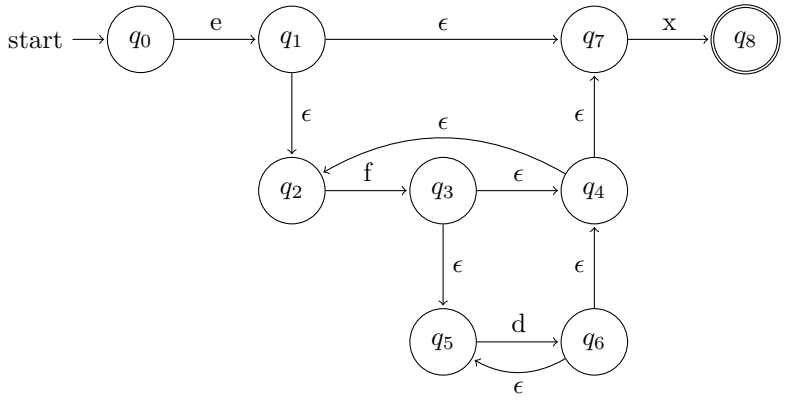
$$ex,\ efx,\ effx,\ efdx,\ efffdx,\ efddfddfx$$

hvor disse strenge ville være ugyldige:

$$fx,\ fdx,\ efd,\ edx$$

Spørgsmål 1.2

Jeg har i dette spørgsmål konstrueret en NFA ved hjælp af en systematisk konstruktion som vist i undervisningen og i *Converting Reguler Expressions to Discrete Finite Automata: A tutorial* af David Christiansen



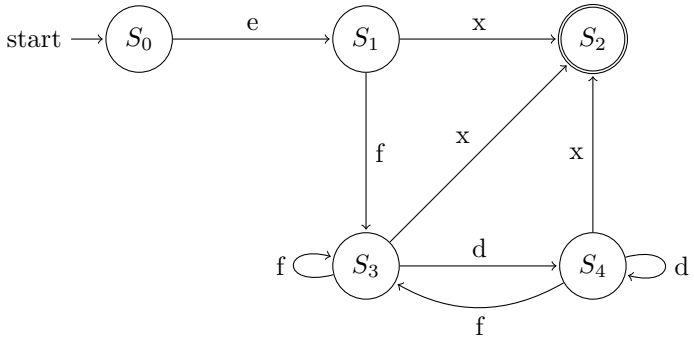
Jacob Benjamin Cholewa jbec@itu.dk

Spørgsmål 1.3

Jeg har i denne opgave brugt en systematisk tilgang til at oversætte min NFA vist i spørgsmål 2. Det første jeg gør er at prøve at fjerne alle ϵ overgange. Det kan jeg opnå ved at lave en DFA hvor hver stadie svare til et sæt stadier fra den gamle NFA. Et nyt stadie i DFA'en svare til et stadie og alle de stadier som kan nås med ϵ overgange fra det første stadie. I tabellen neden for viser jeg de nye DFA stadier, hvilke overgange de har til andre stadier og hvilke gamle NFA stadier de svare til.

State	e	f	d	x	NFA States
S_0	S_1	-	-	-	$\{q_0\}$
S_1	-	S_3	-	S_2	$\{q_1, q_2, q_7\}$
S_2	-	-	-	-	$\{q_8\}$
S_3	-	S_3	S_4	S_2	$\{q_2, q_3, q_4, q_5, q_7\}$
S_4	-	S_3	S_4	S_2	$\{q_2, q_4, q_5, q_6, q_7\}$

Derefter har jeg tegnet en DFA svarende til tabellen



Spørgsmål 1.4

Jeg er lidt i tvivl om hvordan dette sprog skal tolkes. Jeg går derfor i opgaven ud fra at man skal *fetch beer* før man kan *drink beer* og *spill beer*. Jeg går ud fra at man kun må *fetch beer* **én** gang da det ikke vises i nogle af eksemplerne at der hentes mere end en øl. Ved denne forståelse kan man danne dette regulære udtryk for sproget:

$$f[sd]*$$

Jacob Benjamin Cholewa jbec@itu.dk

Opgave 2

Spørgsmål 2.1

I denne opgave har jeg implementeret `Pair, Fst` og `Snd` til den abstrakte syntaks for HigherFun sproget. Dette har jeg gjort ved at ændre i `absyn.fs` og udvide `expr` typen. Der vises her kun et udsnit af `expr` typen.

```
type expr =
  | Pair of expr * expr
  | Fst of expr
  | Snd of expr
  ...
```

Spørgsmål 2.2

I denne opgave bygger jeg videre på opgave 2.1 og udvider nu `HigherFun.fs` til at kunne bruge den udvidede abstrakte syntaks. Når et udtryk evalueres i vores lille funktionelle sprog returneres en værdi med typen `value`. Jeg har nu udvidet `value` typen med `ValuePair`, altså et par af værdier, så vi kan evaluere det nye `Pair` udtryk

```
type value =
  | Int of int
  | ValuePair of value * value
  | Closure of string * string * expr * value env
```

Derefter har jeg ændret i evaluerings logikken så der nu kan evalueres udtryk med `Pair, Fst` og `Snd`. Der vises her kun et lille del af evaluerings logikken. Som det kan ses evaluerer vi `Pair` ved både at evaluere `e1` og `e2` til et `ValuePair`. Jeg bruger *Pattern Matching* til at evaluere `Fst` og `Snd` udtryk og sørger samtidig for at det kun er den del af parret man bruger der bliver evalueret. Hvis `Fst` og `Snd` bliver kaldt med andet end et `Pair` som indre udtryk vil de ikke blive *matched* og programmet vil fejle med beskeden *Invalid syntax*

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    ...
    | Pair(e1,e2)   -> ValuePair((eval e1 env),(eval e2 env))
    | Fst(Pair(e1,e2)) -> eval e1 env
    | Snd(Pair(e1,e2)) -> eval e2 env
    ...
    | _               -> failwith "Invalid syntax"
```

Jeg tolker i denne opgave "Bemærk at `Fst` og `Snd` skal fejle, hvis værdien de anvendes på ikke er et parudtryk." som at jeg kun skal kunne tolke udtryk som

```
let ex = Snd(Pair (CstI 1,Pair (CstI 2,CstI 3)))
```

og ikke

```
let ex = Fst(Snd(Pair (CstI 1,Pair (CstI 2,CstI 3))))
```

Jeg forklare i Spørgsmål 2.3 hvad det betyder for funktionaliteten og hvordan man kunne have implementeret `eval` så den kunne evaluere udtryk som det sidste

Spørgsmål 2.3

Jeg har brugt et lille hjælpe program `run` som kan findes i appendiks. Filen `ex23` som indeholder mine eksempler kan også findes i appendiks A.

```
$ ./run < ex23
  ...
> val ex1 : expr = Pair (CstI 1,Pair (CstI 2,CstI 3))
> val ex2 : expr = Fst (Pair (CstI 1,Pair (CstI 2,CstI 3)))
> val ex3 : expr = Snd (Pair (CstI 1,Pair (CstI 2,CstI 3)))
> val ex4 : expr =
  Let
    ("x",CstI 3,
     Pair
       (Letfun ("f","x",Prim ("*",Var "x",CstI 2),Call (Var "f",Var "x")),
        Letfun ("f","x",Prim ("+",Var "x",CstI 2),Call (Var "f",Var "x"))))

> val res1 : value = ValuePair (Int 1,ValuePair (Int 2,Int 3))
> val res2 : value = Int 1
> val res3 : value = ValuePair (Int 2,Int 3)
> val res4 : value = ValuePair (Int 6,Int 5)
```

Min implementation fungere, men et valg jeg træf var at gøre så `Fst` og `Snd` kun evaluere den del af parret som der er brug for. Det gør så også at min implementation ikke kan evaluere udtryk som disse:

```
let ex5 = Snd(Snd(ex1));;
let ex6 = Fst(ex4);;
```

Ved at implementere metoden på denne måde kunne man have evalueret udtryk som `ex5` og `ex6`, men så bliver man nødt til at evaluere hele parret hvilke ikke gøres i den nuværende implementation.

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    ...
    | Pair(e1,e2)   -> ValuePair((eval e1 env),(eval e2 env))
    | Fst(e) ->  match eval e env with
                 | ValuePair(v1,v2) -> v1
                 | _                -> failwith "not a pair"
    | Snd(e) ->  match eval e env with
                 | ValuePair(v1,v2) -> v2
                 | _                -> failwith "not a pair"
    ...
    | _               -> failwith "Invalid syntax"
```

Spørgsmål 2.4

I denne opgave har jeg udvidet parser og lexer specifikationen til at indeholde de tre nye udtryk `Pair, Fst` og `Snd` med syntaksen

```
(1,true)
```

```
fst(1,true)
snd(1,true)
```

Det første jeg har gjort er at tilføje de nye keywords og token. Parenteser er allerede lagt ind som tokens.

```
let keyword s =
    match s with
    ...
    | "fst"  -> FST
    | "snd"  -> SND
    | _      -> NAME s

...

rule Token = parse
  ...
  | ','           { COMMA }
  ...
```

Derefter udvider jeg parser specifikationen.

```
Expr:
  ...
  | LPAR Expr COMMA Expr RPAR       { Pair($2,$4)          }
  | FST Expr                        { Fst($2)              }
  | SND Expr                        { Snd($2)              }
  ...
;
```

Jeg har testet at implementationen virker med følgende udtryk

```
run (fromString "(1,true)");;
run (fromString "fst(1,true)");;
run (fromString "snd(1,true)");;
```

der evaluere til

```
$ ./run < ex24
  ...
> val it : HigherFun.value = ValuePair (Int 1,Int 1)
> val it : HigherFun.value = Int 1
> val it : HigherFun.value = Int 1
```

Filen `ex24` kan ses i appendiks A.

Spørgsmål 2.5

Jeg har omskrevet mine eksempler fra opgave 2.3 til

```
(1,(2,3))
fst(1,(2,3))
snd(1,(2,3))
let x = 3 in ((let f x = x * 2 in f x end), (let f x = x + 2 in f x end)) end
```

og har derefter evalueret dem

```
$ ./run < ex25
  ...
> val it : HigherFun.value = ValuePair (Int 1,ValuePair (Int 2,Int 3))
> val it : HigherFun.value = Int 1
> val it : HigherFun.value = ValuePair (Int 2,Int 3)
> val it : HigherFun.value = ValuePair (Int 6,Int 5)
```

Filen `ex25` kan ses i appendiks A.

Spørgsmål 2.6

I denne opgave har jeg udledt et typeinferenstræ for udtrykket

$$snd(32 < 2, (10 + 2, fst(false, 1 + 3)))$$

Træet er delt op i fire segmenter og refereres med nummeret angivet til højre for segmentet

$$(fst)\frac{(pair)\dfrac{(p2)\,\overline{\rho \vdash false : bool} \qquad (p4)\dfrac{(p1)\,\overline{\rho \vdash 1 : int} \qquad (p1)\,\overline{\rho \vdash 3 : int}}{\rho \vdash 1+3 : int}}{\rho \vdash (false, 1+3) : bool * int}}{\rho \vdash fst(false, 1 + 3) : bool} \tag{1}$$

$$(pair)\frac{(p4)\dfrac{(p1)\,\overline{\rho \vdash 10 : int} \qquad (p1)\,\overline{\rho \vdash 2 : int}}{\rho \vdash 10+2 : int} \qquad (1)}{\rho \vdash (10 + 2, fst(false, 1 + 3)) : int * bool} \tag{2}$$

$$(pair)\frac{(p5)\dfrac{(p1)\,\overline{\rho \vdash 32 : int} \qquad (p1)\,\overline{\rho \vdash 2 : int}}{\rho \vdash 32 < 2 : bool} \qquad (2)}{\rho \vdash (32 < 2, (10 + 2, fst(false, 1 + 3)) : bool * (int * bool)} \tag{3}$$

$$(snd)\frac{(3)}{\rho \vdash snd(32 < 2, (10 + 2, fst(false, 1 + 3))) : int * bool} \tag{4}$$

Jacob Benjamin Cholewa jbec@itu.dk

Opgave 3

Spørgsmål 1

I denne opgave har jeg implementeret kommandoen `ARRLEN` til den abstrakte maskine. Denne kommando tager adressen til et array og lægger længden af arrayet på stakken.

Jeg har for at implementere denne opgave ændret og tilføjet i 5 filer: `Machine.fs, Machine.java, CLex.fsl, CPar.fsy` og `Comp.fs`

Først har jeg tilføjet den nye kommando til `Machine.fs`

```
type instr =
  ...
  | STOP       (* halt the abstract machine    *)
  | ARRLEN     (* get s[sp] = s[sp] - s[s[sp]] *)
...
let CODESTOP  = 25
let CODEARRLEN = 26;
...
let makelabenv (addr, labenv) instr =
    match instr with
    ...
    | STOP          -> (addr+1, labenv)
    | ARRLEN        -> (addr+1, labenv)

let rec emitints getlab instr ints =
    match instr with
    | Label lab     -> ints
    ...
    | STOP          -> CODESTOP  :: ints
    | ARRLEN        -> CODEINDEX :: ints
```

Nu da kommandoen er tilføjet til den abstrakte maskines syntaks skal vi ændre den abstrakte maskine til at kunne udfører kommandoen. Elementet på et arrays adresse a, som lægger lige efter arrayets elementer, er en pointer til arrayets første element på adresse q. Der ved kan man udregne arrayets længde ud da $arrlen = a - q$.

Før `ARRLEN` kommandoen kaldes lægges array adressen på stakken, og arrayets længde kan så udregnes ved

$$\begin{aligned} a &= s[sp] \\ q &= s[s[sp]] \\ arrlen &= a - q \end{aligned}$$

Denne logik er implementeret i den abstrakte maskine.

```
final static int
  ...
  STOP = 25,
```


Jacob Benjamin Cholewa jbec@itu.dk

```
  INDEX = 26;

...

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 ARRLEN:
        s[sp] = s[sp] - s[s[sp]];
        break;
      ...
    }
  }
}

static String insname(int[] p, int pc) {
  switch (p[pc]) {
  ...
  case ARRLEN: return "ARRLEN";
  default:    return "<unknown>";
  }
}
```

Nu skal syntaksen implementeres i parser og lexer specifikationen. Dette gøres ved først at tilføje det nye token til lexer specifikationen

```
rule Token = parse
  ...
  | '|'  { BAR }
  ...
```

hvorefter parser specifikationen rettes til. Jeg implementerer `ARRLEN` som en unary operator for ikke at skulle ændre i den abstrakte syntaks.

```
...
%token BAR
...
ExprNotAccess:
  ...
  | BAR Expr BAR { Prim1("arrlen", $2) }
  ...
;
```

Til sidste implementeres operatoren i Kompileren. `|arr|` parses som en `Access`, men vi har brug for den som en `Addr`, derfor *Pattern matcher* vi adressen og pakker den ind som en `Addr`.

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match e with
    ...
    | Prim1(ope, Access acc) when ope = "arrlen" -> cExpr (Addr acc) varEnv funEnv @
        [ARRLEN]
```

...

Derved har vi implementeret kommandoen `ARRLEN` i den abstrakte maskine og gjort den tilgængelig i MicroC med syntaksen `|arr|`

Spørgsmål 2

I denne opgave har jeg udvidet microC programmet givet i opgaven og testet implementationen af `ARRLEN` fra opgave 3.1.

```
void main(){
        int arr[3];
        arr[0] = 10; arr[1] = 20; arr[2] = 30;
        int sum; sum = 0;
        int i; i = 0;
        while(i < |arr|){
                sum = sum + arr[i];
                i = i + 1;
        }
        print sum;
}
```

Jeg har testet programmet ved hjælp af hjælpe programmet c-run og fik den forventede sum. c-run kan ses i appendiks B

```
$ ./c-run exarrlen
60
Ran 0.0 seconds
```

Spørgsmål 3 og 4

Programmet vist i opgave 3.3 vil ikke virke fordi at b peger på adressen for det første element i arrayet q. Da vores implementation kræver at vi har adressen på elementet i slutningen af arrayet vil dette ikke virke. Denne implementation giver os altså længden mellem det første element i arrayet og pointer b's placering på stakken i stedet for arrayets rigtige længde.

```
./c-run ex33
7
Ran 0.0 seconds
```

Programmet vist i 3.4 virke da vi her laver en pointer til arrayet hvilket altså er en pointer til arrayets sidste element a som vores naive `ARRLEN` implementationen bruger.

```
$ ./c-run ex34
40
Ran 0.0 seconds
```

Opgave 4

Spørgsmål 1

Jeg har i denne opgave implementeret erklæring af nye arrays med syntaksen `int a[b..s..e]` hvor b er den først værdi, s er det der lægges til for hvert trin og e den maksimale værdi.

Jeg har implementeret denne løsning ved at lave en ny type TypR(t,b,s,e) som når den allokeres opretter et nyt array af typen `TypA` og udfylder arrayet med talserien.

```
type typ =
  ...
  | TypA of typ * int option      (* Array type     *)
  | TypR of typ * int * int * int (* Array range type *)
```

Jeg har derefter opdateret lexer specifikationen til at have tokenet `...` Jeg har oprettet det som et samlet token i stedet for et punktum for ikke at tillade mellemrum mellem punktummerne.

```
rule Token = parse
  ...
  | ".."          { DOTDOT }
  ...
```

Derefter har jeg tilføjet det nye token og den nye type til Parser specifikationen. Jeg har tilføjet en ny non terminal for at kunne tillade både positive og negative tal uden at tillade `Boolean` og `NULL`

```
%token BAR DOTDOT
...
Vardesc:
  ...
  | Vardesc LBRACK Number DOTDOT Number DOTDOT Number RBRACK
              { compose1 (fun t -> TypR(t,$3,$5,$7)) $1 }
;
...
Const:
    Number                          { $1      }
  | CSTBOOL                         { $1      }
  | NULL                            { -1      }
;
Number:
    CSTINT                          { $1      }
  | MINUS CSTINT                    { - $2    }
;
```

Til sidste har jeg implementeret logikken i kompileren. Jeg tillader kun at man initialisere int arrays. I stedet for at bruge kommandoen `INCSP i` for at initialisere tomme plader på stakken tilføjer jeg en `CSTI i` kommando for hvert element i rangen og fylder på den måde den nye array op med værdierne fra rangen. Til sidst ligger jeg en reference til det første array element på stakken ligesom ved en normal array initialisering.

```
let allocate (kind : int -> var) (typ, x) (varEnv : varEnv) : varEnv * instr list =
```

```
...
| TypR (t, i1, i2, i3) when t = TypI ->
  if i2 = 0 then failwith "the step paramenter can't be zero"
  let arr = [|i1 .. i2 .. i3|]
  let newEnv = ((x, (kind (fdepth+arr.Length), TypA(t, Some arr.Length))) ::
      env, fdepth+arr.Length+1)
  let code = (Array.fold (fun acc x -> acc @ [CSTI x]) [] arr @ [GETSP; CSTI
      (arr.Length - 1); SUB])
  (newEnv, code)
| TypR (_, _, _, _) ->
  failwith "Only implemented for int arrays"
...
```

Spørgsmål 2 og 3

Jeg har for at kunne teste implementationen skrevet et lille test program der tager en given range, summere den sammen og returnerer resultatet. Jeg har været nødt til at give array længden for arrayet med som parameter da jeg ellers ville få problemet som beskrevet i opgave 3.3

```
void main() {
       int a[1 .. 2 .. 20]; sumArr(a, |a|);
       int b[-100 .. 1 .. 100]; sumArr(b, |b|);
       int c[50 .. -5 .. 0]; sumArr(c, |c|);
       int d[10 .. -2 .. -10]; sumArr(d, |d|);
       int e[10 .. 1 .. 10]; sumArr(e, |e|);
}

void sumArr(int arr[], int len){
       int i; i = 0;
       int sum; sum = 0;
       while (i < len){
              sum = sum + arr[i];
              i=i+1;
       }
       print sum;
}
```

Hvilket evaluere til det forventede resultat.

```
$ ./c-run exrange
100 0 275 0 10
Ran 0.001 seconds
```

A Hjælpe metoder til opgave 2

run

```
#!/bin/bash
make
fsharpi -r FsLexYacc.Runtime.dll Absyn.fs HigherFun.fs FunPar.fs FunLex.fs Parse.fs
    ParseAndRunHigher.fs
```

Makefile

```
all: FunPar.fs FunLex.fs
FunPar.fs: FunPar.fsy
	fsyacc --module FunPar FunPar.fsy
FunLex.fs: FunLex.fsl
	fslex --unicode FunLex.fsl
clean:
	rm -rf FunLex.fs FunPar.fs FunPar.fsi
```

ex23

```
open Absyn;;
open HigherFun;;
let ex1 = Pair(CstI 1, Pair(CstI 2, CstI 3));;
let ex2 = Fst(ex1);;
let ex3 = Snd(ex1);;
let ex4 = Let("x",CstI 3, Pair (Letfun ("f","x",Prim ("*",Var "x",CstI 2),Call (Var
    "f",Var "x")), Letfun ("f","x",Prim ("+",Var "x",CstI 2),Call (Var "f",Var
    "x"))));;
//let ex5 = Snd(Snd(ex1));;
//let ex6 = Fst(ex4);;
let res1 = eval ex1 [];;
let res2 = eval ex2 [];;
let res3 = eval ex3 [];;
let res4 = eval ex4 [];;
//let res5 = eval ex5 [];;
//let res6 = eval ex6 [];;
```

ex24

```
open ParseAndRunHigher;;
run (fromString "(1,true)");;
run (fromString "fst(1,true)");;
run (fromString "snd(1,true)");;
```

ex25

```
open ParseAndRunHigher;;
run (fromString @"(1,(2,3))");;
```

```
run (fromString @"fst(1,(2,3))");;
run (fromString @"snd(1,(2,3))");;
run (fromString @"let x = 3 in ((let f x = x * 2 in f x end), (let f x = x + 2 in f
    x end)) end") ;;
```


Jacob Benjamin Cholewa jbec@itu.dk

B Hjælpe metoder til opgave 3

c-run

```
#!/bin/bash
make > /dev/null
echo \
    'open ParseAndComp;;' \
    'compileToFile (fromFile "'$1'.c") "'$1'.out";;' \
    |fsharpi -r FsLexYacc.Runtime.dll Absyn.fs CPar.fs CLex.fs Parse.fs Machine.fs
        Comp.fs ParseAndComp.fs > /dev/null
java Machine $1.out ${@:2}
```

Makefile

```
all: CLex.fs CPar.fs Machine.class
CLex.fs: CLex.fsl
	fslex --unicode CLex.fsl
CPar.fs: CPar.fsy
	fsyacc --module CPar CPar.fsy
Machine.class: Machine.java
	javac Machine.java
clean:
	rm -rf Machine.class CLex.fs CPar.fs CPar.fsi *.out
```

ex33.c

```
void main(){
	int a[4];
	printlen(a);
}
void printlen(int b[]){
	print |b|;
}
```

ex34.c

```
void main() {
	int a[40];
	printlen(&a);
}
void printlen(int *b[]) {
	print |*b|;
}
```

exarrlen.c

```
void main(){
	int arr[3];
```

```
        arr[0] = 10; arr[1] = 20; arr[2] = 30;
        int sum; sum = 0;
        int i; i = 0;
        while(i < |arr|){
                sum = sum + arr[i];
                i = i + 1;
        }
        print sum;
}
```

exrange.c

```
void main() {
        int a[1 .. 2 .. 20]; sumArr(a, |a|);
        int b[-100 .. 1 .. 100]; sumArr(b, |b|);
        int c[50 .. -5 .. 0]; sumArr(c, |c|);
        int d[10 .. -2 .. -10]; sumArr(d, |d|);
        int e[10 .. 1 .. 10]; sumArr(e, |e|);
}

void sumArr(int arr[], int len){
        int i; i = 0;
        int sum; sum = 0;
        while (i < len){
                sum = sum + arr[i];
                i=i+1;
        }
        print sum;
}
```


Jacob Benjamin Cholewa jbec@itu.dk