Skriftlig eksamen

By Albert Rise Nielsen (albn@itu.dk)

Course Name:	Programmer som data
Course Code:	BSPRDAT1KU
Course Manager:	Niels Hallenberg

IT University of Copenhagen

January 9, 2023

Contents

Indledning

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

1 Opgave 1

1.1 Delopgave 1

1.1.1 Koden

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

let exam1 = Every(Write(numbers));
```

1.1.2 Eksempel

Output fra FSI:

```
> run exam1;;
5 6 7 8 9 10 11 12 val it : value = Int 0
```

1.1.3 Forklaring

Koden skal læses indefra og ud. Så først laves en liste med 8 tal, da `FromTo` er inklusiv. Hvert tal pakkes ind i et `Write` så continuationen nu er en sekvens af 8 `Write` udtryk med et tal i. Hvis koden kørtes nu ville det kun være 5 der printes. For at eksekvere alle udtrykkene pakkes det ind i et `Every` udtryk som eksekverer det hele.

1.2 Delopgave 2

1.2.1 Koden

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

let exam2 = Every(Write(Prim("<", CstI 10, numbers)));
```

1.2.2 Eksempel

Output fra FSI:

```
> run exam2;;
11 12 val it : value = Int 0
```

1.2.3 Forklaring

Koden minder meget om den fra 1.1. Den eneste forskel er at `numbers` pakkes ind i et `Prim` udtryk der sammenligner det med 10 og kun printer tallene hvis de er over 10. Da 10 tallet placeres først i sammenligningen så er det $10 < 5, 10 < 6, \ldots$ der sammenlignes, hvilket betyder at det kun er tallene højere end 10 der evaluerer sandt.

1.3 Delopgave 3

1.3.1 Koden

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

let exam3 = Every(Write(Prim("<", numbers, And(Write (CstS "\n"), numbers))));
```

1.3.2 Eksempel

Output fra FSI:

```
> run exam3;;

 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
```

1.3.3 Forklaring

Koden er en kombination af de tidligere opgaver. I det andet argument af `Prim` gives nu sekvensen. Mens i det tredje argument gives et `And` udtryk som evaluerer begge af sine argumenter men kun returnerer det andet. Så her printer koden først en ny linje med ny linje karakteren `\n`, derefter returnerer den `numbers` sekvensen som vi kender den. Selve sammenligningen bliver nu til $8 \cdot 8$ sammenligninger, da vi sammenligner alle elementer i den sidste sekvens med 5, printer en ny linje, sammenligner med 6 osv til og med 12. Dermed får vi en sekvens af tal hvor der fjernes et ved hver iteration. 5 vises aldrig da $5 < 5$ ikke er sandt, det sidste 12 tal er sammenligningen $11 < 12$, så der er en sidste tom linje der bare ikke vises. En dårlig ting ved strategien er den ekstra linje i toppen af outputtet, og at `val it : ...` vises på en ny linje. Den første er fordi `Write` evalueres før sekvensen, den sidste er fordi er er en tom linje.

1.4 Delopgave 4

1.4.1 Koden

Ændringer i `Icon.fs`

```
type expr =
  ...
  | FromToChar of char * char
  | Fail;;

let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    ...
    | FromToChar(c1, c2) ->
      let rec loop c =
        if c <= c2 then
          cont (Str (c |> string)) (fun () -> loop ((c |> int) + 1 |> char))
        else
          econt ()
```

```
15
16      loop c1
17    | Fail -> econt ()
18
19 let chars = FromToChar('C', 'L');
20 let exam4 = Every(Write(chars));
```

1.4.2 Eksempel

Output fra FSI:

```
1  > run exam4;;
2  C D E F G H I J K L val it : value = Int 0
3
4  > run chars;;
5  val it : value = Str "C"
6
7  > run (FromToChar('D', 'A'));;
8  Failed
9  val it : value = Int 0
10
11 > run (Every(Write(FromToChar('D', 'c'))));;
12 D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c val it : value = Int 0
```

1.4.3 Forklaring

Koden er baseret på `FromTo` generatoren, samme løkke mekanik kan bruges da `char` typen kan sammenlignes ligesom en `int` i FSharp. Dog skal der lidt konvertering til at gå op i tegntabellen. Det gøres ved at oversætte `char` til `int - c |> int`, der adderes 1 og derefter konverteres tilbage til `char - ...+ 1 |> char`. Se eksempel herunder. En del af opgaven var at generatoren laver `Str` typen, som tager en `string`. Det kan opnås ved at konveretere `char` til `string` således `c |> string`, som så pakkes ind i `Str` typen.

```
1 > ('D' |> int) + 1 |> char;;
2 val it : char = 'E'
```

En ting der er værd at notere er at store og små bogstaver ikke ligger lige efter hinanden i tegntabellen så man kan ikke generere små til store, kun store til små, og hvis man gør så får man tegnene `[ ] ^ _ ``` imellem tegnene.

1.5 Delopgave 5

1.5.1 Koden

```
1  let rec eval (e : expr) (cont : cont) (econt : econt) =
2      match e with
3      ...
4      | Prim(ope, e1, e2) ->
5        eval e1 (fun v1 -> fun econt1 ->
6            eval e2 (fun v2 -> fun econt2 ->
7                match (ope, v1, v2) with
8                ...
9                | ("<", Str s1, Str s2) ->
10                   if s1<s2 then
11                       cont (Str s2) econt2
12                   else
13                       econt2 ()
```

```
14
15              | _ -> Str "unknown prim2")
16              econt1)
17          econt
18    ...
19
20 let exam5_1 = Prim("<", CstS "A", CstS "B");
21 let exam5_2 = Prim("<", CstS "B", CstS "A");
```

1.5.2 Eksempel

Output fra FSI:

```
1 > run exam5_1;;
2 val it : value = Str "B"
3
4 > run exam5_2;;
5 Failed
6 val it : value = Int 0
```

1.5.3 Forklaring

Løsningen er ret simpel, det er en kopi af sammenligningen af `Int` ændret til at tage `Str` i stedet. I FSharp er strenge sammenlignlige ligesom integers så der skal ikke ændres meget.

1.6 Delopgave 6

1.6.1 Koden

```
1 let chars = FromToChar('C', 'L');
2
3 let exam6 = Every(Write(Prim("<", CstS "G", chars)));
```

1.6.2 Eksempel

Output fra FSI:

```
1 > run exam6;;
2 H I J K L val it : value = Int 0
```

1.6.3 Forklaring

Løsningen er en simpel afart af 1.2. Istedet for integers anvendes den nye sammenligning der understøtter strenge og så skal vi bare have alle tegn over `'G'`.

2 Opgave 2

2.1 Delopgave 1

2.1.1 Koden

Kode ændringer givet her i diff format.

```
diff --git a/exercise-2/CLex.fsl b/exercise-2/CLex.fsl
index 52e30e0..dc15b8b 100644
--- a/exercise-2/CLex.fsl
+++ b/exercise-2/CLex.fsl
@@ -35,6 +35,10 @@ let keyword s =
     | "true"    -> CSTBOOL 1
     | "void"    -> VOID
     | "while"   -> WHILE
+    | "createStack" -> CREATESTACK
+    | "pushStack" -> PUSHSTACK
+    | "popStack" -> POPSTACK
+    | "printStack" -> PRINTSTACK
     | _         -> NAME s

 let cEscape s =
diff --git a/exercise-2/CPar.fsy b/exercise-2/CPar.fsy
index ed3a85f..3d6d73d 100644
--- a/exercise-2/CPar.fsy
+++ b/exercise-2/CPar.fsy
@@ -14,7 +14,7 @@ let nl = CstI 10
 %token <int> CSTINT CSTBOOL
 %token <string> CSTSTRING NAME

-%token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE
+%token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE CREATESTACK PUSHSTACK
    POPSTACK PRINTSTACK
 %token NIL CONS CAR CDR DYNAMIC SETCAR SETCDR
 %token PLUS MINUS TIMES DIV MOD
 %token EQ NE GT LT GE LE
@@ -134,6 +134,10 @@ ExprNotAccess:
   | Expr LE    Expr                    { Prim2("<=", $1, $3)   }
   | Expr SEQAND Expr                   { Andalso($1, $3)       }
   | Expr SEQOR  Expr                   { Orelse($1, $3)        }
+  | CREATESTACK LPAR Expr RPAR         { Prim1("createStack", $3) }
+  | PUSHSTACK LPAR Expr COMMA Expr RPAR { Prim2("pushStack", $3, $5) }
+  | POPSTACK LPAR Expr RPAR            { Prim1("popStack", $3) }
+  | PRINTSTACK LPAR Expr RPAR          { Prim1("printStack", $3) }
 ;

 AtExprNotAccess:
```

2.1.2 Eksempel

```
# mono 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");
```

```
9           Stmt (Expr (Assign (AccVar "s", Prim1 ("createStack", CstI 3))));
10          Stmt (Expr (Prim2 ("pushStack", Access (AccVar "s"), CstI 42)));
11          Stmt (Expr (Prim2 ("pushStack", Access (AccVar "s"), CstI 43)));
12          Stmt (Expr (Prim1 ("printStack", Access (AccVar "s"))));
13          Stmt (Expr (Prim1 ("printi", Prim1 ("popStack", Access (AccVar "s")))));
14          Stmt (Expr (Prim1 ("printi", Prim1 ("popStack", Access (AccVar "s")))));
15          Stmt (Expr (Prim1 ("printStack", Access (AccVar "s"))))])]
```

2.1.3 Forklaring

Der skal en meget lille ændring til for at parse funktionerne. Først laves de som symboler i lexeren (`CLex.fsl`), derefter registreres de i parseren (`CPar.fsy`). Også i parser filen defineres deres format og output. Først defineres formatet ved at bruge symbolet, parentes symbolet (`LPAR`), udtryk specifikationen (`Expr`), og derefter luk parantes (`RPAR`). Herefter defineres output, i form af `Prim1` som gives en streng der kan matches på senere og det parsede argument. Det gentages for hver funktion, funktioner med 2 argumenter bruger komma symbolet (`COMMA`) mellem hvert argument og bruger `Prim2` istedet for `Prim1`.

2.2 Delopgave 2

2.2.1 Koden

Machine.fs:

```
type instr =
  ...
  | CREATESTACK                         (* create stack                    *)
  | PUSHSTACK                           (* push to top of stack            *)
  | POPSTACK                            (* pop top of stack                *)
  | PRINTSTACK                          (* print stack                     *)

...
let CODESETCDR = 31
let CODECREATESTACK = 32
let CODEPUSHSTACK   = 33
let CODEPOPSTACK    = 34
let CODEPRINTSTACK  = 35;


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

listmachine.c:

```
...
#define STACKTAG 1
...

#define CREATESTACK 32
#define PUSHSTACK 33
#define POPSTACK 34
#define PRINTSTACK 35

void printInstruction(word p[], word pc) {
  switch (p[pc]) {
```

```
  ...
  case CREATESTACK:
    printf("CREATESTACK");
    break;
  case PUSHSTACK:
    printf("PUSHSTACK");
    break;
  case POPSTACK:
    printf("POPSTACK");
    break;
  case PRINTSTACK:
    printf("PRINTSTACK");
    break;
  ...
}

int execcode(word p[], word s[], word iargs[], int iargc,
             int /* boolean */ trace) {
  ...
    case CREATESTACK: {
      word n = Untag(s[sp]);

      if (n < 0) {
        printf("Cannot create a negative sized stack\n");
        return -1;
      }

      word *p = allocate(STACKTAG, n + 3, s, sp);
      s[sp] = (word)p; // Insert header

      p[1] = Tag(n); // Size
      p[2] = Tag(0); // Top
    } break;
    case PUSHSTACK: {
      word p = s[sp - 1];

      if (p == 0) {
        printf("Cannot push to null\n");
        return -1;
      }

      word *stack = (word *)p;
      word size = Untag(stack[1]);
      word top = Untag(stack[2]);

      if (top >= size) {
        printf("Cannot push to full stack\n");
        return -1;
      }

      word v = s[sp];
      stack[top + 3] = v;
      stack[2] = Tag(top + 1);
      sp--;
    } break;
    case POPSTACK: {
      word p = s[sp];

      if (p == 0) {
        printf("Cannot push to null\n");
        return -1;
      }
```

```
74
75        word *stack = (word *)p;
76        word top = Untag(stack[2]);
77
78        if (top == 0) {
79          printf("Cannot pop from empty stack");
80          return -1;
81        }
82
83        s[sp] = stack[top + 2];
84        stack[2] = Tag(top - 1);
85      } break;
86      case PRINTSTACK: {
87        word p = s[sp];
88        word *stack = (word *)p;
89
90        word n = Untag(stack[1]);
91        word top = Untag(stack[2]);
92        printf("STACK(" WORD_FMT ", " WORD_FMT ")=[ ", n, top);
93        for (int i = 0; i < top; i++) {
94          printf(WORD_FMT " ", Untag(stack[i + 3]));
95        }
96        printf("]\n");
97      } break;
98      ...
99  }
```

2.2.2 Eksempel

Se 2.3

2.2.3 Forklaring

I `Machine.fs` defineres instruktionerne med de definerede navne, derefter defineres deres instruktions nummer, de starter ved 32 og ender med 35. Til sidst defineres label miljøet. Her sker der ikke noget specielt da instruktionerne selv ikke tager nogen argumenter, de tages fra stakken.

`listmachine.c` starter også med at definere instruktionerne og deres nummer, samt et tag til stakke. `printInstruction` udvides med en streng version af instruktions navnet. Det spændende er `execcode`, her defineres håndteringen af instruktionerne.

Først defineres `CREATESTACK`. Den finder størrelsen N og allokerer hukkomelsen til stakken. Der allokeres som specifikationen siger, 3 ord mere end den adspurgte størrelse. Heri gemmes en header, størrelsen N og hvor mange elementer der er indsat *top*. En pointer til stakken placeres på stakken.

`PUSHSTACK` henter stakken fra stakken og henter det element der skal pushes fra stakken. Derefter hentes top, og det nye element indsættes på stakken. *top* øges med 1.

`POPSTACK` henter stakken fra stakken og henter *top*. Derefter hentes det øverste element fra stakken og placeres på stakken. *top* nedsættes med 1. En ting der er værd at notere er at elementet hentes ved $top + 2$ istedet for $top + 3$ som normalt, da *top* er 1 foran det rent faktiske element.

`PRINTSTACK` henter stakken fra stakken og henter størrelsen og top. Derefter printes stakken ved at køre igennem fra $header\_lokation + 3$ til $header\_lokation + (top - 1)$. -1 af samme grund som $+2$ i `POPSTACK`.

2.3 Delopgave 3

2.3.1 Koden

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match e with
    ...
    | Prim1(ope, e1) ->
      cExpr e1 varEnv funEnv
      @ (match ope with
         ...
         | "createStack" -> [CREATESTACK]
         | "popStack" -> [POPSTACK]
         | "printStack" -> [PRINTSTACK]

    | Prim2(ope, e1, e2) ->
      cExpr e1 varEnv funEnv
      @ cExpr e2 varEnv funEnv
      @ (match ope with
         ...
         | "pushStack" -> [PUSHSTACK]
```

2.3.2 Eksempel

Output fra FSI:

```
# mono listcc.exe stack.lc
List-C compiler v 1.0.0.0 of 2012-02-13
Compiling stack.lc to stack.out
Prog
  [Fundec
  ...

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

Used 0 cpu milli-seconds
```

2.3.3 Forklaring

Super simpel ændring. Instruktioner med 1 argument placeres i `Prim1` matches og mapper bare til sin egen instruktion. Instruktioner med 2 argumernter placeres ligeså i `Prim2`.

2.4 Delopgave 4

2.4.1 Forklaring

I forhold til tests så burde der være tests for success og fejl. Her er det altså når `createStack` får en negativ N værdi, hvis `pushStack` får en 0 værdi som pointer eller hvis der ikke er plads på stakken. Det successfulde testcase er allerede lavet i `stack.lc`, så her kommer fejlene.

2.4.2 Koden

`test1.lc`:

```
void main() {
  dynamic s;
  s = createStack(-1); // Fails here
}
```

test2.lc:

```
void main() {
  dynamic s;
  s = createStack(2);
  pushStack(s,42);
  pushStack(s,43);
  pushStack(s,43); // Fails here
  printStack(s);
}
```

test3.lc:

```
void main() {
  dynamic s;
  pushStack(s,42); // Fails here
}
```

2.4.3 Eksempel

```
# mono listcc.exe test1.lc
# mono listcc.exe test2.lc
# mono listcc.exe test3.lc

# ./ListVM/ListVM/listmachine test1.out
Cannot create a negative sized stack

Used 0 cpu milli-seconds

# ./ListVM/ListVM/listmachine test2.out
Cannot push to full stack

Used 0 cpu milli-seconds

# ./ListVM/ListVM/listmachine test3.out
Cannot push to null

Used 0 cpu milli-seconds
```

3 Opgave 3

3.1 Delopgave 1

3.1.1 Koden

```
type typ =
  ...
  | TypT of typ * int option          (* Tuple type                  *)

and access =
  ...
  | TupIndex of access * expr         (* Tuple indexing              *)
```

3.1.2 Eksempel

Output fra FSI:

```
> 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)
```

3.1.3 Forklaring

De to tilføjelser er lavet efter opgavens meget detaljerede beskrivelse.

3.2 Delopgave 2

3.2.1 Koden

```
rule Token = parse
  ...
  | "(|"              { LPARBAR }
  | "|)"              { RPARBAR }
  | '('               { LPAR }
```

3.2.2 Forklaring

Tilføjelsen er lavet efter opgavens meget detaljerede beskrivelse.

3.3 Delopgave 3

3.3.1 Koden

```
Vardesc:
  ...
  | Vardesc LPARBAR RPARBAR             { compose1 (fun t -> TypT(t, None)) $1    }
  | Vardesc LPARBAR CSTINT RPARBAR      { compose1 (fun t -> TypT(t, Some $3)) $1 }
;

Access:
  ...
  | Access LPARBAR Expr RPARBAR         { TupIndex($1, $3)    }
```

3.3.2 Kompilering

Jeg kompilerer med en række scripts som jeg har skrevet. De kan ses herunder.

compile.sh

```
#!/bin/sh
binDir=/data/bin

$binDir/compileLexer.sh C
$binDir/compileParser.sh C

$binDir/run.sh Absyn.fs CPar.fs CLex.fs Parse.fs Interp.fs ParseAndRun.fs Machine.fs Comp
    .fs ParseAndComp.fs
```

compileLexer.sh

```
#!/bin/sh
binDir=/data/bin

mono $binDir/fslex.exe --unicode $1Lex.fsl
```

compileParser.sh

```
#!/bin/sh
binDir=/data/bin

mono $binDir/fsyacc.exe --module $1Par $1Par.fsy
```

run.sh

```
#!/bin/sh
binDir=/data/bin

fsharpi -r $binDir/FsLexYacc.Runtime.dll $@
```

3.3.3 Eksempel

```
# ./compile.sh
...

> 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")])]

> 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)))))])]
```

3.3.4 Forklaring

Parser tilføjelserne er lavet efter opgavens meget detaljerede beskrivelse og er baseret på liste implementationen.

3.4 Delopgave 4

3.4.1 Koden

```
let allocate (kind : int -> var) (typ, x) (varEnv : varEnv) : varEnv * instr list =
    let (env, fdepth) = varEnv
    match typ with
    ...
    | TypA (TypT _, _) ->
      raise (Failure "allocate: array of tuples not permitted")
    ...
    | TypT (TypT _, _) ->
      raise (Failure "allocate: tuple of tuples not permitted")
    | TypT (TypA _, _) ->
      raise (Failure "allocate: tuple of arrays not permitted")
    | TypT (t, Some i) ->
      let newEnv = ((x, (kind (fdepth), typ)) :: env, fdepth+i)
      let code = [INCSP i;]
      (newEnv, code)

and cAccess access varEnv funEnv : instr list =
    match access with
    ...
    | TupIndex(acc, idx) -> cAccess acc varEnv funEnv
                            @ cExpr idx varEnv funEnv @ [ADD]
```

3.4.2 Eksempel

```
# ./compile.sh
...

> open ParseAndComp;;
> compile "tuple";;
...
> #quit;;

```

```
# java Machine tuple.out
55 56 55 56
Ran 0.0 seconds
```

3.4.3 Forklaring

Løsningen er, som resten af opgaven, baseret på liste implementationen. Første tilføjelse er at `TypA` ikke kan instantieres med `TypT` elementer. De næste to tilføjelser er samme ide, `TypT` kan ikke instantieres med sin egen type eller `TypA`. Det er ikke understøttet da der bruges ret simpel indeksering ind i hukkomelsen som ikke ville kunne håndtere at skulle bruge et offset for hver liste i liste.

Den sidste tilføjelse er instantiering af `TypT` med en konstant størrelse. Den skiller sig ud fra liste implementationen da der ikke er en pointer til det første element. Så variablen der tilføjes til miljøet er lokationen af det første element, i modsætning til en pointer til det første element. Derfor er instantierings instruktionerne også ret simple, da de bare udvider stakken med størrelsen. Tilføjelsen i `cAccess` reflekterer den ændring ved ikke at lave en `LDI` efter `cAccess` evalueringen. Det er fordi resultatet af `cAccess` er lokationen af det første element, så den følgende addition af indekset er direkte derpå.

3.5 Delopgave 5

3.5.1 Forklaring

Ændringerne er forklaret i de opgaver hvor de redigeres i.

3.6 Delopgave 6

3.6.1 Forklaring

Eksemplet fra 3.4 duplikeres her.

3.6.2 Eksempel

```
# ./compile.sh
...

> open ParseAndComp;;
> compile "tuple";;
...
> #quit;;

# java Machine tuple.out
55 56 55 56
Ran 0.0 seconds
```

4 Opgave 4

4.1 Delopgave

4.1.1 Koden

FunLex.fsl

```
rule Token = parse
  ...
  | '['              { LBRACK }
  | ']'              { RBRACK }
  | ','              { COMMA }
  | '@'              { AT }
```

FunPar.fsy

```
...
\%token PLUS MINUS TIMES DIV MOD AT
\%token LPAR RPAR LBRACK RBRACK
\%token COMMA
...

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

Expr:
  ...
  | Expr PLUS  Expr                    { Prim("+",  $1, $3)     }
  | Expr AT    Expr                    { Prim("@",  $1, $3)     }
  ...
;

AtExpr:
  ...
  | LBRACK List RBRACK                 { List($2)               }
;


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

Absyn.fs

```
type expr =
  ...
  | List of expr list
```

4.1.2 Eksempel

Output fra FSI:

```
> open Parse;;
> fromFile "ex01.txt";;
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])))

> fromFile "ex02.txt";;
System.Exception: parse error in file ex02.txt near line 1, column 10
...

> fromFile "ex03.txt";;
val it : Absyn.expr =
  Let
    ("l", List [CstI 43],
     Prim ("@", Var "l", List [Prim ("+", CstI 3, CstI 4)]))

> fromFile "ex04.txt";;
val it : Absyn.expr =
  Let
    ("l", List [CstI 3],
     Prim
       ("=", Prim ("@", Var "l", List [CstI 3]),
        List [Prim ("+", CstI 3, CstI 4)]))

> fromFile "ex05.txt";;
val it : Absyn.expr =
  Letfun ("f", "x", Prim ("+", Var "x", CstI 1), List [Var "f"])

> fromFile "ex06.txt";;
val it : Absyn.expr = Letfun ("id", "x", Var "x", List [Var "id"])
```

4.1.3 Forklaring

Koden handler bare om at registrere symbolerne [], @ og så parse dem som lister. For at gøre det laves et nyt parsing koncept `List`, som rekursivt parser liste elementer, hvor hvert element er adskilt af et komma. Den pakkes ind i vores firkantede parentes symboler for at fuldene parsing af listen.

ex02 fejler fordi opgaven specifikt siger "En ikke tom liste", så den kan ikke parse den tomme liste.

4.2 Delopgave

4.2.1 Koden

```
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 =
    match e with
    ...
    | List l -> ListV (List.map (fun e -> eval e env) l)
```

```
    | Prim(ope, e1, e2) ->
      let v1 = eval e1 env
      let v2 = eval e2 env
      match (ope, v1, v2) with
      ...
      | ("=", ListV l1, ListV l2) -> Int (if l1 = l2 then 1 else 0)
      | ("@", ListV l1, ListV l2) -> ListV (l1 @ l2)
      ...
    ...
```

4.2.2 Eksempel

Output fra FSI:

```
> open ParseAndRunHigher;;
> open Parse;;
> run(fromFile "ex01.txt");;
val it : HigherFun.value = Int 1

> run(fromFile "ex02.txt");;
System.Exception: parse error in file ex02.txt near line 1, column 10
...

> run(fromFile "ex03.txt");;
val it : HigherFun.value = ListV [Int 43; Int 7]

> run(fromFile "ex04.txt");;
val it : HigherFun.value = Int 0

> run(fromFile "ex05.txt");;
val it : HigherFun.value =
  ListV [Closure ("f", "x", Prim ("+", Var "x", CstI 1), [])]

> run(fromFile "ex06.txt");;
val it : HigherFun.value = ListV [Closure ("id", "x", Var "x", [])]
```

4.2.3 Forklaring

Først tilføjes typen til `value` som opgaven siger. Derefter tilføjes en konstruktør i `eval`, den mapper expressions for at evaluere til deres `value` form. I `Prim` tilføjes et `=` case, som bare bruger FSharps indbyggede lighed på lister, da den opfylder specifikationen. Til sidst tilføjes et `@` primitiv som også bruger den interne version af sig selv.

4.3 Delopgave 3

Se figur 1

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

Figur 1: Typetræ for ex03.txt