IT UNIVERSITY OF COPENHAGEN

Eksamensaflevering i Programmer som data, Januar 2023

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

Student information

Name	Email
Andreas Wachs	ahja@itu.dk

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

Indhold

1. Introduktion

Dette dokument indeholder min besvarrelse på denne eksamensopgave.

Dokumentet indeholder mange kode-snippets for at dokumentere mine løsninger på opgaverne. Dog har jeg taget den frihed at udelade kildekode som ikke er relevant, eller var til givet. Det betyder at jeg indsætter `(...)` med passende kommentarsymboler foran for at vise at jeg undlader at vise dele af kildekoden som jeg ikke har ændret.

En kopi af kursets repository med mine ændrniger vil kunne findes på min personlige GitHub profil. Det kopierede repository offentliggøres efter endt afleveringsfrist på denne opgave (10. Januar kl 14:00).
Se mere: `https://github.com/andreaswachs/PRDAT-exam`.

Jeg har arbejdet på disse opgaver på en Linux maskine ved brug af Mono. Dette betyder at du kan komme til at se at jeg benytter binære filer som `fsharpi` og `fsharpc` modsat Microsoft's .NET værktøjer.

2. Icon

Dette kapitel gør brug af filer fundet i kursets repository under mappen `Lectures/Lec11/Cont/`. Det kan antages af filer nævnt i dette kapitel gør brug af filer under denne mappe eller andre undermapper.

2.1 Icon opgave 1

For at løse denne opgave, skal jeg bruge funktionerne `Every` og `Write`, samt bruge det gemte udtryk `numbers`:

```
> 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
```

Her bruger jeg `Every` til at gennemgå alle de mulige løsninger som gives af generatoren `numbers`. `Write` bruges til at skrive tallene ("løsningerne") ud på skærmen.

2.2 Icon opgave 2

For at løse denne opgave skal jeg genbruge sidste opgaves løsning samt gøre brug af `Prim` udtrykket:

```
> Every(Write(Prim("<", CstI 10, numbers)));;
val it : expr = Every (Write (Prim ("<", CstI 10, FromTo (5, 12))))

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

Her gennemløbes tallene 5 til 12, hvor vi kan "filtrere"på tallene fra generatoren `numbers` vha. `Prim` udtrykket med `<` operatoren, sådan at der kun udskrives 11 og 12 på skærmen.

2.3 Icon opgave 3

```
> run (Every(Write(Or(Or(Prim("<", numbers, numbers), numbers), CstS "\n"))));;
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 5 6 7 8 9 10
     11 12
 val it : value = Int 0
```

Dette program har jeg ikke implementeret korrekt. Jeg kan se at jeg producere serier af tal som bliver gradvist kortere efter hver serie. Men programmet stopper ikke på det rigtige tidspunkt, eller producerer linjeskift som i opgaveteksten.

2.4 Icon opgave 4

Jeg starter med at udvide den abstrakte syntax i `Icon.fs`:

```
type expr =
  | CstI of int
  | CstS of string
  | FromTo of int * int
  | FromToChar of char * char // Icon opgave 4
  | Write of expr
  | If of expr * expr * expr
  | Prim of string * expr * expr
  | And of expr * expr
  | Or  of expr * expr
  | Seq of expr * expr
  | Every of expr
  | Fail;;
```

Dernæst udvider jeg `eval` funktionen til at kunne håndtere `FromToChar` udtryk:

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    // (...)
    | FromToChar(c1, c2) -> // Icon opgave 4
      let rec loop c1' =
        if c1' <= c2 then
          cont (Str (string c1')) (fun () -> loop (c1' + char 1))
        else
          econt ()
      loop c1
    // (...)
```

Denne implementation virker på at `char` er en primitiv datatype. Det gælder for `char` at man kan lave aritmetik (fx. `'A' + (char 1) = 'B'`) og sammenligninger med værdier af denne type. Herved kan jeg køre programmet som angivet i opgaveteksten:

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

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

2.5 Icon opgave 5

Jeg udvider `eval` funktionen i førnævnte `Icon.fs` fil, under det indlejrede `match` udtryk for operatoren for et `Prim` udtryk:

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    // (...)
    | Prim(ope, e1, e2) ->
      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) -> // Icon opgave 5
10                  if (char s1) < (char s2) then
11                      cont (Str s2) econt2
12                  else
13                      econt2 ()
14              // (...)
```

Her laves de to strenge om til `char`s, som kan sammenlignes. Jeg kører dette i `fsharpi`:

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

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

Jeg ser også at man kan få implementationen til at selvdestruere, hvis man ikke indsætter en gyldig `char` som string:

```
> run (Prim("<", CstS "Aaa", CstS "B"));;
System.FormatException: String must be exactly one character long.
  at System.Char.Parse (System.String s) [0x00017] in (...)
```

Dette skyldes at `"Aaa"` ikke er en gyldig `char` fordi at strengen indeholder mere end en karakter.

2.6 Icon opgave 6

For at udskrive bokstaverne `H` til og med `L` vil jeg gøre brug af `Prim`, som nu understøtter sammenligning mellem bokstaver:

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

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

Dette program virker ved at vi gennemtvinger og udskriver alle løsninger med `Every` og `Write` funktionerne. For at finde de enkelte løsninger i programmet bruges `Prim` funktionen med `<` operatoren til kun at frakaste alle løsninger hvor det generede bokstav fra `chars`-generatoren har en lave leksikalsk værdi end `G`.

3. ListC

I dette kapittel implementeres en hoballokeret stak til ListC sproget.

Jeg gør brug af Makefiles til at gemme lange kommandoer, som gentages I løbet af opgaven. Du kan se indholdet af den pågældende Makefile for dette kapitel i appendix A.1.

Jeg arbejder med filerne som findes i kursets repository under mappen `Lectures/Lec10/ListC`. Alle filer som henvises til i det følgende kapitel findes i denne mappe eller en undermappe.

3.1 Tilpasning af parser- og lexerspecifikationerne

For at implementere stakke i List-C sproget, starter jeg med at udvide den abstrakte syntaks og parseren.
Jeg starter med at definere nye *keywords* til lexerspecifikationen (`CLex.fsl`) og definerer navne for tilhørende tokens:

```
let keyword s =
    match s with
    // (...)
    | "createStack" -> CREATESTACK
    | "pushStack"   -> PUSHSTACK
    | "popStack"    -> POPSTACK
    | "printStack"  -> PRINTSTACK
    // (...)
```

Jeg tilføjer de nye tokens i parseren (`CPar.fsy`):

```
%token CREATESTACK PUSHSTACK POPSTACK PRINTSTACK
```

Samt laver jeg parserspecifikationen for de gældende tokens:

```
AtExprNotAccess:
  // (...)
  | 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) }
;
```

For at validere mine ændringer, ændrer jeg `FromFile` funktionen i `Parse.fs` filen til at udskrive det abstrakte syntaxtræ ved oversættelse.
Jeg gemmer programmet fra opgaveteksten til filen `test1.lc`:

3.2 Udvidelse af byte code instruktioner

3.2.1 Machine.fs

Jeg starter med at udvide `Machine.fs` for at kunne oversætte symbolsk byte kode til numerisk repræsenteret byte kode.
Jeg udvider instruktions typen:

```
type instr =
  // (...)
  | CREATESTACK                     (* creates a new stack of size N*)
  | PUSHSTACK                       (* Pushes value $snd into stack pointed to at $fst *)
  | POPSTACK                        (* Pops from stack at address int *)
  | PRINTSTACK                      (* Prints the stack at address int *)
```

Jeg tilføjer til listen over numeriske byte koder for de nye instruktioner:

```
let CODECREATESTACK = 32;
let CODEPUSHSTACK   = 33;
let CODEPOPSTACK    = 34;
let CODEPRINTSTACK  = 35;
```

Jeg udvider `makelabenv` funktionen:

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

og til sidst udvider jeg `emitints` funktionen i `Machine.fs` filen:

```
let rec emitints getlab instr ints =
    match instr with
    // (...)
    | CREATESTACK -> CODECREATESTACK :: ints
    | PUSHSTACK   -> CODEPUSHSTACK :: ints
    | POPSTACK    -> CODEPOPSTACK :: ints
    | PRINTSTACK  -> CODEPRINTSTACK :: ints
```

Dette udgør ændringerne for `Machine.fs` filen. Som det næste udvider jeg `listmachine.c` filen.
Nu hvor `Machine.fs` er færdig, kan jeg undersøge om oversættelsen foregår korrekt:
Jeg gemmer programmet fra opgavebeskrivelsen:

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

```
8     print popStack(s);
9     printStack(s);
10 }
```

Dernæst bygger jeg lexer- og parserspecifikationerne og oversætterprogrammet `listcc.exe`. Jeg ændrer `Parse.fs` til at udskrive det abstrakte syntakstræ under oversættelsen, hvorefter jeg oversætter det ovenstående program:

```
1 $ make build
2 $ mono listcc.exe test1.lc
3 List-C compiler v 1.0.0.0 of 2012-02-13
4 Compiling test1.lc to test1.out
5 Prog
6   [Fundec
7      (None, "main", [],
8       Block
9         [Dec (TypD, "s");
10         Stmt (Expr (Assign (AccVar "s", Prim1 ("createStack", CstI 3))));
11         Stmt (Expr (Prim2 ("pushStack", Access (AccVar "s"), CstI 42)));
12         Stmt (Expr (Prim2 ("pushStack", Access (AccVar "s"), CstI 43)));
13         Stmt (Expr (Prim1 ("printStack", Access (AccVar "s"))));
14         Stmt (Expr (Prim1 ("printi", Prim1 ("popStack", Access (AccVar "s")))));
15         Stmt (Expr (Prim1 ("printi", Prim1 ("popStack", Access (AccVar "s")))));
16         Stmt (Expr (Prim1 ("printStack", Access (AccVar "s"))))])]
```

Endvidere kan jeg se at der er produceret byte kode i filen `test1.out`:

```
1 $ cat test1.out
2 24 19 0 5 25 26 13 0 0 1 0 3 32 12 15 -1 13 0 0 1 (...)
```

3.2.2 listmachine.c

De fire funktioner for at operere med vores hob allokeret stak (`createStack, pushStack, popStack, printStack`) vil nu blive implementeret.

Alle funktionerne kræver at håndtere de forskellige byte koder der angiver et kald til hver sin funktion. Alle ændringerne er samlet i `execcode` funktionen i `listmachine.c` filen.

Alle kodestykker vil indeholde forklarende kommentarer, der forklarerer hvad der foregår og hvorfor. Jeg vil supplere en kort opsummering efter hvert kodestykke.
Nedenfor ses implementationen til at håndtere `createStack` funktionen:

```
1 case CREATESTACK: {
2   // Aflæs den ønskede stak størrelse
3   word stack_size = Untag((word) s[sp]);
4
5   // Fejlhåndtering for størrelser på stakke som er negative
6   // Til diskussion: Kan en stak på størrelse 0 overhovedet bruges til noget spændende
      ?
```

```
7   if (stack_size < 0) {
8     printf("Cannot allocate stack with negative capacity (%d)\n", stack_size);
9     return -1;
10  }
11
12  // Alloker hukommelse til stakken, hvor vi skal huske at have plads til de første
13  // tre pladser til header, N og top
14  word* p = allocate(STACKTAG, stack_size + 3, s, sp);
15
16  // Nu "prepper" vi stakken
17  p[1] = Tag(stack_size); // N
18  p[2] = Tag(0); // top
19
20  // Sætter alle værdierne på stakken til 0, og vi husker at tagge
21  for (int i = 0; i < stack_size; i++) {
22    p[3+i] = Tag(0);
23  }
24
25  // Opdater program stakken, erstat størrelsen med en pointer til stakken
26  s[sp] = (word) p;
27 } break;
```

Implementationen af `createStack` starter ved at aflæse det øverset element på selve programmets stak. Dette er parameteret, som angiver hvor stor den hoballokerede stak skal være.
Der foretages et check for at forhindre forsøg på at skabe stakke af negative størrlser. Derefter allokeres plads på hoben til at opbevare hele stakken, som er de antal ønksede elementer samt de 3 "administrative"felter: `header`, `N` og `top`, som findes på hukommelsespladserne 0, 1 og 2 på den hoballokerede staks hukommelsesområde. `N` gemmes til at være lig størrelsen på stakken og `top` sættes til at være 0.
Derefter gennemgås stakkens område for opbevaring af værdier, hvor alle værdierne sættes til `0`.
Til sidst opdateres program stakken. Det øverste element på stakken udskiftes fra at være størrelsen på hoben til at opbevare en pointer til den nye hoballokerde stak, jfr appendix A.2.

Efterfølgende findes implementationen for funktionen `pushStack`:

```
1  case PUSHSTACK: {
2    // Læs værdien v og addresse pointeren p
3    word v = Untag((word) s[sp]);
4    word* p = (word*) s[--sp];
5
6    // Aflæs stakkens størrelse og "offsettet" på det øverste element
7    word size = Untag(p[1]);
8    word top = Untag(p[2]);
9
10   if (top == size) {
11     printf("Could not push to a full stack! Size: %d", size);
12     return -1;
13   }
14
```

```
15   // Sæt det nye topelement to værdien v
16   p[++top + 2] = Tag(v);
17
18   // Opdater "offsettet" til top elementet
19   p[2] = Tag(top);
20 } break;
```

Her indlæses værdien v og addressen p, jfr. skemaet for byte koderne i appendix A.2. Dernæst læser vi stakkens totale størrelse `size` og placeringen af toppen af stakken `top`.
For at indsætte en værdi på den hoballokerede stak undersøger vi først om stakken er fuld. Der undersøgers hvorvidt `top == size`, og stopper udførelsen af programmet hvis stakken er fuld.
I tilfældet hvor stakken ikke er fuld fortsættes programmet. Her indsættes væriden v på stakken ved at bruge `top` til at bestemme det rette index. Derefter forøges og gemmes værdien for `top` med 1.

Dernæst er det tid til at implementere `popStack` funktionen, som fjerner og returnerer det øverste element fra den hoballokerede stak.
Min implementation er som følgende:

```
1  case POPSTACK: {
2    // Indsæt pointeren til vores heap stak
3    word* p = (word*) s[sp];
4
5    // aflæs "offsettet" til det øverste element
6    word top = Untag(p[2]);
7
8    // Check to see if the stack is empty before popping
9    if (top < 1) {
10     printf("Attempted to pop from an empty stack!\n");
11     return -1;
12   }
13
14   // Aflæs det øverste element fra stakken, hvor vi bruger offset plus
15   // et offset på 2 for at læse forbi vores "bogholder"-værdier (index [0-2])
16   word v = Untag((word) p[top + 2]);
17   s[sp] = v;
18
19   // Juster offsettet til det øverste element, siden at vi har fjernet
20   // en værdi fra stakken
21   p[2] = Tag(--top);
22 } break;
```

Først aflæses der en pointer til den hoballokerede stak fra program stakken. Dernæst aflæses `top`, som angiver antal elementer tilstede i vores stak.
Skulle der være færre end ét element i stakken, vil programmets afvikling standes pga. at det er en fejl at forsøge at fjerne elementer fra en tom stak.
I tilfældet hvor der er flere end 0 elementer på stakken, vil programmet aflæse det øverste element og trække 1 fra `top` og opdatere `top`. Værdien som blev aflæst overskriver nu det øverste

element i program stakken, som var pointeren til den hoballokerede stak.

For at kunne undersøge den hoballokerede staks udvikling under afvikling af programmer, skal jeg implementere den sidste funktion: `printStack`:

```
case PRINTSTACK: {
  // Aflæs pointeren til vores heap stak
  word* p = (word*) s[sp];

  // aflæs dets totale størrelse og offsettet
  word size = Untag(p[1]);
  word top =  Untag(p[2]);

  // Begynd at skrive stak "tracen" med total størrelse og antal elementer i det pt.
  printf("STACK(%d, %d)=[ ", size, top);

  // Udskrive r alle værdierne på vores heap stak
  for (int i = 0; i < top; i++) {
    printf("%d ", Untag(p[3+i]));
  }
  printf("]\n"); // Lukker formatteringen hvorend brugeren læser det
} break;
```

`printStack` har til mening udelukket at fortælle om den hoballokerede stak's tilstand. Derfor forventer den at en pointer til førnævnte stak ligger øverst på program stakken.
Vha. pointeren aflæser den stakkens allokerede størrelse `size` (`N`) og antal tilstedeværende elementer `top`.
Herefter udskriver den disse to værdier, samt udskrives alle elementer som er til stede i stakken. Her er det interessant at elementer som tilføjes og fjernes fra stakken kan stadig være til stede i hukommelsen, da hukommelsen ikke bliver nulstillet efter et `popStack` kald. Men ved at bruge `top` værdien så ser vi ingen af de tidligere fjernede elementer.

3.2.3 Test af listmachine

Her vil jeg beskrive de vigtigste test cases for at afdække hvorvidt funktionerne `createStack` og `pushStack` er implementeret korrekt.

De vigtigste test cases for at afdække `createStack`'s korrekthed er at teste for hvorvidt hoballokerede stakke kan skabes for størrelser der er gyldige og ugyldige. Nedenfor findes der 2 test cases for netop dette:

`stack_create_stack_1.lc`

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

`stack_create_stack_3.lc`

```
void main() {
    dynamic s;
    s = createStack(-2);
    printStack(s);
}
```

Ved at bruge `listmachine` kan disse 3 test cases checkes:

```
$ mono listcc.exe stack_create_stack_1.lc
$ ./listmachine stack_create_stack_1.out
STACK(2, 0)=[ ]

Used 0 cpu milli-seconds

$ mono listcc.exe stack_create_stack_2.lc
$ ./listmachine stack_create_2.out
Cannot allocate stack with negative capacity (-2)

Used 0 cpu milli-seconds
```

Herved kan jeg bekræfte at implementationen virker ved at funktionen stopper programafvikling, hvis der bedes om en hoballokeret stak med negativ størrelse.
De vigtigste testcases for at finde ud af om implementationen for `pushStack` er korrekt er at afdække hvorvidt man kan tilføje værdier til en hob som ikke er fuld, og en som er fuld.
Følgende test cases for at dække disse to scenaier er som fælgende:

`stack_push_stack_1.lc`:

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

`stack_push_stack_2.lc`:

```
void main() {
    dynamic s;
    s = createStack(1);
    printStack(s);
    pushStack(s, 42);
    printStack(s);
    pushStack(s, 43);
    printStack(s);
}
```

Ved at eksekvere de følgende to test cases, kan det følgende output ses:

```
$ mono listcc.exe stack_push_stack_1.lc
# (...)
$ ./listmachine stack_push_stack_1.out
STACK(2, 0)=[ ]
STACK(2, 1)=[ 42 ]

Used 0 cpu milli-seconds

$ mono listcc.exe stack_push_stack_2.lc
# (...)
$ ./listmachine stack_push_stack_2.out
STACK(1, 0)=[ ]
STACK(1, 1)=[ 42 ]
Could not push to a full stack! Size: 1
Used 0 cpu milli-seconds
```

Her ser vi at test casen for at tilføje værdien 42 til en tom stak med størrelse 2 gennemføres med success.
I test casen hvor værdierne 42 og 43 prøves at tilføjes til en stak med total størrelse 1, vil programmet fejle ved forsøg på at indsætte værdien 43.

4. MicroC

I dette kapittel implementerer jeg tuples i MicroC sproget, set under forelæsning 6.

Der gøres en grundantagelse om at jeg bruger filer som ligger i kursets kode repository under mappen `Lectures/Lec06/MicroC`. Filnavne som nævnes i løbet af kapitlet stammer fra denne mappe.

Makefil benyttet i dette kapitel kan findes i appendix B.1.

4.1 Udvidelse af den abstrakte syntaks

For at kunne repræsentere typen for tupler skal den abstrakte syntaks udvides. Dette sker i `Absyn.fs`, hvor jeg laver følgende ændringer:

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

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

Her tilføjer jeg den nye datatype `TypT`, som repræsenterer tupler. Tupler er en mængde med en given data type og muligvis elementer, repræsenteret af `option` typen(Wlaschin 2012). Derudover udvides gøres det muligt at repræsentere adgang til elementer i en tuple i det abstrakte syntakstræ ved at udvide `access` med `TupIndex` som tager en `access` for at finde den pågældende tuple, som er allokeret på stakken og et udtryk `expr` som skal evalueres til et index i tuplen.
Jeg påviser at denne ændringer virker ved at åbne en interaktiv session af fsharpi og instantierer typer af disse to ændringer:

```
$ make fsi
# (...)

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

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

Her ses det at jeg får samme resultat som opgavebeskrivelsens eksempel. Derudover kan jeg også påvise at udvidelsen af `access` er jfr. opgavebeskrivelsens eksempel:

```
# (...)

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

4.2 Udvidelse af lexer- og parserspecifikationen

Tuples i MicroC vil blive repræsenteret ved brug af de specielle "klammer"`(|` og `|)`. For at allokere en tuple skal man derfor skrive `int a(|3|);` for at allokere en tuple med heltal, med størrelsen 3. For at sætte et af værdierne i en tuple vil man udtrykke `a(|0|) = 10;`, hvor element 0 i tuplen vil blive tildelt værdien 10. For at aflæse elementet igen vil man skrive `a(|0|);`.

Derfor skal specifikationerne for lexeren og parseren udvides.

Jeg starter med at udvide lexeren til at genkende disse klammer og angiver to nye tokens:

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

Her introduceres to nye tokens `LPARBAR` og `RPARBAR`. Disse skal skrives ind i parseren, hvor der ogås skal laves regler for hvordan at tuple dannelse og adgang skal parses.
Jeg tilføjer de nye tokens:

```
%token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE
%token PLUS MINUS TIMES DIV MOD
%token EQ NE GT LT GE LE
%token NOT SEQOR SEQAND
%token LPARBAR RPARBAR // De nye tokens
%token LPAR RPAR LBRACE RBRACE LBRACK RBRACK SEMI COMMA ASSIGN AMP
%token EOF

%nonassoc LPARBAR // Per hint i opgaveteksten.
```

Her tilføjes de som tokens, samt gøres den venstre klamme ikke associativ for at undgå shift/reduce problemer.

Som det næste udvider jeg parseren til at kunne erklære nye tupler:

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

Her ses det at man både kan lave tomme tupler samt tupler med en given længde.

Som det næste udvides parseren med parsing af `access` til tuple elementer:

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

Jeg vil nu påvise at mine udvidelser producerer samme resultater som i opgavens eksempel:

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

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

Her ses det at udvidelsen i lexer- og parserspecifikationen gør at oversætteren nu er i stand til at oversætte nye erklæringer af tuples, henholdsvis 2 og ingen elementer.
For at påvise at udtrykket `t1(|e|)` kan parses som L-værdi, vil jeg køre kode eksemplerne fra opgaveteksten:

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

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

Her ses det at der kan bygges abstrakte syntakstræer for programmer hvor en tuple `t1` får sat det nulte element til værdien 55 og programemr hvor et givent element, her det nulte, bliver læst fra en tuple.

4.3 Udvidelse af oversætteren

Indtil videre har jeg udvidet dele af MicroC oversætteren sådan at MicroC kildekode kan omdannes til et abstrakt syntakstræ. Nu skal oversætteren `Comp.fs` udvides, sådan at de nye udvidelser kan oversættes til byte kode. Alle ændringer laves i `Comp.fs`.

Jeg starter med at udvide `allocate` funktionen, som generer byte kode for allokering af data på program stakken:

```
let allocate (kind : int -> var) (typ, x) (varEnv : varEnv) : varEnv * instr list =
    // (...)
    | TypT (_, Some i) ->
      let newEnv = ((x, (kind (fdepth + i), typ)) :: env, fdepth+i)
      let code = [INCSP i]

      (newEnv, code)
     | _ -> // (...)
```

Her skabes der et opdateret variabel miljø, hvor vi har defineret den nye tuple. Her angives størrelsen af tuplen, som kun omfatter antallet af elementer i tuplen. Modsat *arrays* bogfører denne tuple ikke for hvor mange elementer der er i tuplen ved at indsætte størrelsen for enden af tuplen.

Byte koden som genereres er udelukket for at lave plads til tuplens elementer. Der gøres intet for at initialisere elementerne til bestemte værdier.
Den anden og sidste del af udvidelsen af oversætteren sker ved for at muliggøre udgang til elemetnerne i tuples:

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

Her udvides funktionen `cAccess` til at håndtere tilgang til tuple elemneter. `acc` henviser til adressen hvor tuplens første element befinder sig på stakken. Ved at bruge funktionen `cAccess` kan vi bestemem denne lokation. Dog bliver der tillagt base pegerens værdi til den lokation, så jeg har valgt at trække det fra lige efter ved `@ [GETBP; SUB]`. Dette er ikke den mest optimale løsning, men det ærligt talt det jeg kunne få til at virke i den tid jeg havde.
Dernæst beregnes det index `idx`, som der ønskes adgang til ved udtrykket `cExpr idx varEnv funEnv`. De to værdier lægges til vha. `ADD` instruktionen, og efterlade lokationen på stakken for det ønskede element i tuplen.

4.4 Kørsel af tuple programemt

For at påvise at min implementation virker, vil jeg oversætte det følgende MicroC program fra opgaveteksten:

```
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) {
```

```
10      print t1(|i|); // 55 56
11      i = i + 1;
12    }
13  }
```

Her erklæres der en tuple af størrelse 2 med typen `int`. Deræfter indsættes to tal: 55 og 56. Disse to tal udskrives efter hvert tal er blevet sat ind. Til sidst løbes tuplen igennem vha. en løkke, og tallene udskrives igen.

Jeg starter med at gemme koden til filen `tupple.c`. Dernæst genererer jeg byte kode vha. oversætteren:

```
1  $ make fsi
2  (...)
3
4  > open ParseAndComp;;
5  > compile "tupple";;
6  val it : Machine.instr list =
7    [LDARGS; CALL (0, "L1"); STOP; Label "L1"; INCSP 2; GETBP; CSTI 2; ADD;
8     GETBP; SUB; CSTI 0; ADD; CSTI 55; STI; INCSP -1; GETBP; CSTI 2; ADD; GETBP;
9     SUB; CSTI 0; ADD; LDI; PRINTI; INCSP -1; GETBP; CSTI 2; ADD; GETBP; SUB;
10    CSTI 1; ADD; CSTI 56; STI; INCSP -1; GETBP; CSTI 2; ADD; GETBP; SUB; CSTI 1;
11    ADD; LDI; PRINTI; INCSP -1; INCSP 1; GETBP; CSTI 2; ADD; CSTI 0; STI;
12    INCSP -1; GOTO "L3"; Label "L2"; GETBP; CSTI 2; ADD; GETBP; SUB; GETBP;
13    CSTI 2; ADD; LDI; ADD; LDI; PRINTI; INCSP -1; GETBP; CSTI 2; ADD; GETBP;
14    CSTI 2; ADD; LDI; CSTI 1; ADD; STI; INCSP -1; INCSP 0; Label "L3"; GETBP;
15    CSTI 2; ADD; LDI; CSTI 2; LT; IFNZRO "L2"; INCSP -3; RET -1]
```

Dernæst bruger jeg den virtuelle Java maskine til at afvikle byte koden:

```
1  java Machine tupple.out
2  55 56 55 56
3  Ran 0.01 seconds
```

Her kan jeg se at udskriften af programmet er lig med eksemplet fra opgaveteksten, udover at kørselstiden er forskellig. Et udskriv hvor jeg kører programmet med `Machinetrace`, der udskriver stakkens tilstand for hver byte kode instruktion kan ses i appendix B.2.

5. MicroML

I dette kaptitel vil jeg implementere lister samt operatører til at lægge lister sammen og bedømme om to er ens.

Jeg gør brug af `make` værktøjet til at gøre det nemmere at bygge lexer- og parserspecifikationen samt fortolkeren for MicroML. Kilden til dette kapitels `Makefile` findes i appendix
Jeg gør brug af kildekode i mappen `Lectures/Lec05` fra kursets repository. Alle filer nænvn i dette kapitel kan formodnes at befinde sig under denne filsti.

5.1 Udvidelse af abstrakt syntax

For at kunne repræsentere lister under parsing skal den abstrakte syntaks udvides. Jeg laver følgende tilføjelse i `Absyn.fs`:

```
type expr =
  // (...)
  | List of expr list
```

Her tiløjes en data type som repræsentere lister og indeholder en list af udtryk (`expr`).

5.2 Udvidelse af lexer- og parserspecifikationerne

Som det næste udvider jeg lexerspecifikationen. Her har jeg brug for at kunne danne 4 symboler om til tokens: `","`, `"@"`, `"["`, `"]"`. Listerne implementeres med elementer som er kommasepareret. Listerne omgives af kantede parenteser og der laves en token for operationen for at lægge lister sammen.

```
rule Token = parse
  // (...)
  | ','             { COMMA }
  | "@"             { LISTAPPEND }
  | '['             { LBRACKET }
  | ']'             { RBRACKET }
  // (...)
```

Herved har jeg skabt koblingen mellem de 4 nye symboler og 4 nye tokens.

Jeg udvider parserspecifikationen ved at håndtere disse nye tokens. Først angiver jeg dem som tokens og angiver precedens for den venstre kantede parentes (`LBRACKET`):

```
%token ELSE END FALSE IF IN LET NOT THEN TRUE COMMA // NY: COMMA
%token PLUS MINUS TIMES DIV MOD LISTAPPEND // NY: LISTAPPEND
```

```
%token EQ NE GT LT GE LE
%token LPAR RPAR LBRACKET RBRACKET // NY: LBRACKET BRACKET
%token EOF
```

Nu er de nye tokens angivet sådan at parseren kan arbejde med dem. Som det næste skal jeg angive hvordan at parseren skal parse udtryk der danner lister, samt udtryk som sætter lister sammen:

```
Expr:
  // (...)
  | Expr LISTAPPEND Expr                { Prim("@", $1, $3)      }
  | List                                { $1 }
;

ListElement:
    Expr                                { [$1]                   }
    | ListElement COMMA Expr            { $1 @ [$3]               }
;

List:
    LBRACKET ListElement RBRACKET       { List($2)               }
;
```

Her er parserspecifikationen angivet for både at parse lister og sammenlægningen af lister vha. `@` operatoren.

Det gælder for parsing af lister at en liste starter med en kantet parentes, indeholder en eller flere komma-separeret udtryk og sluttes af med en kantet parentes igen. `List` udtrykket beskriver listens overordnede form, hvor `ListElement` parser rekursivt udtryk inden i listen.

For at parse et udtryk som lægger to lister sammen, gælder det at disse lister kan være en liste som er angivet direkte i kildekoden, fx: `[1] @ [2]`, samt kan det være tilfældet at listen kommer fra et udtryk, fx: `let l = [1] in l @ [2] end`. Dette betyder at for at parse disse operationer skal parseren tillade `Expr` rundt om `@` symbolet.

5.2.1 Test af ændringer

For at teste om hvorvidt MicroML kode med lister kan forstås af lexer og parseren, vil jeg køre programmerne `ex01` til `ex06` som er angivet i opgaveteksten.

```
$ make fsi
(...)

> open Parse;;
> printfn "### ex01 ###"; fromString "let l1 = [2, 3] in let l2 = [1, 4] in l1 @ l2 =
    [2, 3, 1, 4] end en- d";;
### ex01 ###
val it : Absyn.expr =
  Let
```

```
9     ("l1", List [CstI 2; CstI 3],
10     Let
11       ("l2", List [CstI 1; CstI 4],
12        Prim
13          ("=", Prim ("@", Var "l1", Var "l2"),
14           List [CstI 2; CstI 3; CstI 1; CstI 4])))
15
16 > printfn "### ex02 ###"; fromString "let l = [] in l end";;
17 ### ex02 ###
18 System.Exception: parse error near line 1, column 10
19
20   at Microsoft.FSharp.Core.PrintfModule+PrintFormatToStringThenFail@1433[TResult].
       Invoke (System.String message) [0x00000] in <3c7b3c57d9e482c47eea39d081a6cc48>:0
21   at FSI_0002.Parse.fromString (System.String str) [0x0009a] in <422
       aa3d409d7478688d60ae304544202>:0
22   at <StartupCode$FSI_0017>.$FSI_0017.main@ () [0x0001f] in <422
       aa3d409d7478688d60ae304544202>:0
23   at (wrapper managed-to-native) System.Reflection.RuntimeMethodInfo.InternalInvoke(
       System.Reflection.RuntimeMethodInfo,object,object[],System.Exception&)
24   at System.Reflection.RuntimeMethodInfo.Invoke (System.Object obj, System.Reflection.
       BindingFlags invokeAttr, System.Reflection.Binder binder, System.Object[]
       parameters, System.Globalization.CultureInfo culture) [0x0006a] in <
       de882a77e7c14f8ba5d298093dde82b2>:0
25 Stopped due to error
26 > printfn "### ex03 ###"; fromString "let l = [43] in l @ [3+4] end";;
27 ### ex03 ###
28 val it : Absyn.expr =
29   Let
30     ("l", List [CstI 43],
31      Prim ("@", Var "l", List [Prim ("+", CstI 3, CstI 4)]))
32
33 > printfn "### ex04 ###"; fromString "let l = [3] in l @ [3] = [3+4] end";;
34 ### ex04 ###
35 val it : Absyn.expr =
36   Let
37     ("l", List [CstI 3],
38      Prim
39        ("=", Prim ("@", Var "l", List [CstI 3]),
40         List [Prim ("+", CstI 3, CstI 4)]))
41
42 > printfn "### ex05 ###"; fromString "let f x = x+1 in [f] end";;
43 ### ex05 ###
44 val it : Absyn.expr =
45   Letfun ("f", "x", Prim ("+", Var "x", CstI 1), List [Var "f"])
46
47 > printfn "### ex06 ###"; fromString "let id x = x in [id] end";;
48 ### ex06 ###
49 val it : Absyn.expr = Letfun ("id", "x", Var "x", List [Var "id"])
```

Her ser jeg at program `ex02` fejler, da tomme lister ikke er understøttet, hvilket er i overensstemmelse med opgaveteksten. De andre programmer producerer det forventede abstrakte syntakstræ.

5.3 Udvidelse af fortolkeren

De sidste ændringer i koden foregår i fortolkeren `HigherFun.fs`, hvor funktionaliteten for `@` og `=` operatoren for lister implementeres i funktionen `eval`. `eval` udvides også til at kunne fortolke udtryk med lister.
Nedenfor vises alle ændringerne i `HigherFun.fs`:

```
type value =
  // (...)
  | ListV of value list (* support evaluation of functions *)

let rec eval (e : expr) (env : value env) : value =
    match e with
    // (...)
    | Prim(ope, e1, e2) ->
      let v1 = eval e1 env
      let v2 = eval e2 env
      match (ope, v1, v2) with
      // (...)
      | ("=", ListV lst1, ListV lst2) -> Int (if lst1 = lst2 then 1 else 0)
      | ("@", ListV lst1, ListV lst2) -> ListV (lst1 @ lst2)
      // (...)
    | List(lst) ->
      // Evaluer alle udtrykkene i listen i miljet env
      let lstElements = List.map (fun e -> eval e env) lst
      ListV (lstElements)
      // (...)
```

Her udvides typen `value` med `ListV`, som er fortoklerens representation af lister, som indeholder en liste af `value`'s.
Derudover implementeres funktionaliteten for operationerne `@` og `=` på lister. F# understøtter de samme operationer for lister, så dette gøres kort ved brug af de samme operationer. Dog giver F operationen `=` en boolsk værdi tilbage, så vi er nød til at omdatte det til `1` eller `0`, da MicroML ikke understøtter bolske værdier.
Til sidst udvides fortolkeren til at oversætte den liste som vi fik med fra parsing skridtet, til fortolkerens egen list `ListV`. Listen fra den abstrakte syntax indeholder en liste af udtryk, som kan evalueres i miljøet `env`. Her bruger jeg F#'s `List.map` til at omdanne alle udtryk i listen til fortolkerens `value` datatype og returnere `ListV (listElements)`.

5.3.1 Test af fortolkeren

For at påvise at ændringerne til fortolkeren tillader at den kan udføre programmer med lister, vil jeg nu køre programmerne `ex01` til `ex06` fra opgavebeskrivelsen:

```
$ make fsi-higher
(...)

> open ParseAndRunHigher;;
> printfn "### ex01 ###"; run <| fromString "let l1 = [2, 3] in let l2 = [1, 4] in l1
    @ l2 = [2, 3, 1, 4] end end";;
### ex01 ###
val it : HigherFun.value = Int 1

> printfn "### ex02 ###"; run <| fromString "let l = [] in l end";;
### ex02 ###
System.Exception: parse error near line 1, column 10

  at Microsoft.FSharp.Core.PrintfModule+PrintFormatToStringThenFail@1433[TResult].
    Invoke (System.String message) [0x00000] in <3c7b3c57d9e482c47eea39d081a6cc48>:0
  at FSI_0002.Parse.fromString (System.String str) [0x0009a] in <
    be027c7583874b08a2e83034108266fd>:0
  at FSI_0002.ParseAndRunHigher+fromString@7-4.Invoke (System.String str) [0x00000] in
     <be027c7583874b08a2e83034108266fd>:0
  at <StartupCode$FSI_0011>.$FSI_0011.main@ () [0x00029] in <
    be027c7583874b08a2e83034108266fd>:0
  at (wrapper managed-to-native) System.Reflection.RuntimeMethodInfo.InternalInvoke(
    System.Reflection.RuntimeMethodInfo,object,object[],System.Exception&)
  at System.Reflection.RuntimeMethodInfo.Invoke (System.Object obj, System.Reflection.
    BindingFlags invokeAttr, System.Reflection.Binder binder, System.Object[]
    parameters, System.Globalization.CultureInfo culture) [0x0006a] in <
    de882a77e7c14f8ba5d298093dde82b2>:0
Stopped due to error
> printfn "### ex03 ###"; run <| fromString "let l = [43] in l @ [3+4] end";;
### ex03 ###
val it : HigherFun.value = ListV [Int 43; Int 7]

> printfn "### ex04 ###"; run <| fromString "let l = [3] in l @ [3] = [3+4] end";;
### ex04 ###
val it : HigherFun.value = Int 0

> printfn "### ex05 ###"; run <| fromString "let f x = x+1 in [f] end";;
### ex05 ###
val it : HigherFun.value =
  ListV [Closure ("f", "x", Prim ("+", Var "x", CstI 1), [])]

> printfn "### ex06 ###"; run <| fromString "let id x = x in [id] end";;
### ex06 ###
val it : HigherFun.value = ListV [Closure ("id", "x", Var "x", [])]
```

Her ser jeg igen at programmet `ex02` fejler, da tomme lister stadig ikke er understøttet. Programmet `ex03` slutter som forventet ved at returnere en liste `[43, 7]`. Programmet `ex04` slutter som forventet da listerne `[3, 3]` $\neq$ `[7]`.

5.4 Udførelse af typetræ

Nedenfor er et MicroML program med implementation af lister:

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

Jeg vil bestemme typen af programmet ved at udføre et typetræ:

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

Som følge af typetræet, kan jeg se at overstående program typer til `int list`.

Jeg har benyttet mig af aflednignsreglerne fra opgavebeskrivelsen (se appendix C.2) samt reglerne fundet på s. 102 i Programming Language Concepts (Sestoft 2017).

Bibliografi

Mogensen, Torben Æ (2011). *Introduction to compiler design*. Undergraduate topics in computer science. OCLC: ocn753828014. London ; New York: Springer. 204 s. ISBN: 978-0-85729-828-7 978-0-85729-829-4.

Sestoft, Peter (2017). *Programming language concepts*. New York, NY: Springer Berlin Heidelberg. ISBN: 978-3-319-60788-7.

Wlaschin, Scott (7. jun. 2012). *The Option type*. F# for Fun and Profit. URL: `https://fsharpforfunandprofit.com/posts/the-option-type`.

Appendices

A. ListC

A.1 Makefile for ListC

```
.PHONY: build
build:
  fslex --unicode CLex.fsl
  fsyacc --module CPar CPar.fsy
  fsharpc --standalone -r FsLexYacc.Runtime.dll Absyn.fs CPar.fs CLex.fs Parse.fs
    Machine.fs Comp.fs ListCC.fs -o listcc.exe
```

A.2 ListC stak funktioner bytekode tabel

	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.

Figur A.1: Tabel over de nye stakfunktioners bytekode og deres handlign ift. stakkens tilstand. Kopieret fra opgaveteksten.

B. MicroC

B.1 Makefile brugt til opgaven

```
.PHONY: fsi
fsi:
  fslex --unicode CLex.fsl
  fsyacc --module CPar CPar.fsy
  fsharpi -r FsLexYacc.Runtime.dll Absyn.fs CPar.fs CLex.fs Parse.fs Machine.fs Comp.
    fs ParseAndComp.fs
```

B.2 tupple.out stack trace

```
$ java Machinetrace tupple.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 2}
[ 4 -999 0 0 2 2 ]{10: ADD}
[ 4 -999 0 0 4 ]{11: GETBP}
[ 4 -999 0 0 4 2 ]{12: SUB}
[ 4 -999 0 0 2 ]{13: CSTI 0}
[ 4 -999 0 0 2 0 ]{15: ADD}
[ 4 -999 0 0 2 ]{16: CSTI 55}
[ 4 -999 0 0 2 55 ]{18: STI}
[ 4 -999 55 0 55 ]{19: INCSP -1}
[ 4 -999 55 0 ]{21: GETBP}
[ 4 -999 55 0 2 ]{22: CSTI 2}
[ 4 -999 55 0 2 2 ]{24: ADD}
[ 4 -999 55 0 4 ]{25: GETBP}
[ 4 -999 55 0 4 2 ]{26: SUB}
[ 4 -999 55 0 2 ]{27: CSTI 0}
[ 4 -999 55 0 2 0 ]{29: ADD}
[ 4 -999 55 0 2 ]{30: LDI}
[ 4 -999 55 0 55 ]{31: PRINTI}
55 [ 4 -999 55 0 55 ]{32: INCSP -1}
[ 4 -999 55 0 ]{34: GETBP}
[ 4 -999 55 0 2 ]{35: CSTI 2}
[ 4 -999 55 0 2 2 ]{37: ADD}
[ 4 -999 55 0 4 ]{38: GETBP}
[ 4 -999 55 0 4 2 ]{39: SUB}
[ 4 -999 55 0 2 ]{40: CSTI 1}
```

```
[ 4 -999 55 0 2 1 ]{42: ADD}
[ 4 -999 55 0 3 ]{43: CSTI 56}
[ 4 -999 55 0 3 56 ]{45: STI}
[ 4 -999 55 56 56 ]{46: INCSP -1}
[ 4 -999 55 56 ]{48: GETBP}
[ 4 -999 55 56 2 ]{49: CSTI 2}
[ 4 -999 55 56 2 2 ]{51: ADD}
[ 4 -999 55 56 4 ]{52: GETBP}
[ 4 -999 55 56 4 2 ]{53: SUB}
[ 4 -999 55 56 2 ]{54: CSTI 1}
[ 4 -999 55 56 2 1 ]{56: ADD}
[ 4 -999 55 56 3 ]{57: LDI}
[ 4 -999 55 56 56 ]{58: PRINTI}
56 [ 4 -999 55 56 56 ]{59: INCSP -1}
[ 4 -999 55 56 ]{61: INCSP 1}
[ 4 -999 55 56 56 ]{63: GETBP}
[ 4 -999 55 56 56 2 ]{64: CSTI 2}
[ 4 -999 55 56 56 2 2 ]{66: ADD}
[ 4 -999 55 56 56 4 ]{67: CSTI 0}
[ 4 -999 55 56 56 4 0 ]{69: STI}
[ 4 -999 55 56 0 0 ]{70: INCSP -1}
[ 4 -999 55 56 0 ]{72: GOTO 107}
[ 4 -999 55 56 0 ]{107: GETBP}
[ 4 -999 55 56 0 2 ]{108: CSTI 2}
[ 4 -999 55 56 0 2 2 ]{110: ADD}
[ 4 -999 55 56 0 4 ]{111: LDI}
[ 4 -999 55 56 0 0 ]{112: CSTI 2}
[ 4 -999 55 56 0 0 2 ]{114: LT}
[ 4 -999 55 56 0 1 ]{115: IFNZRO 74}
[ 4 -999 55 56 0 ]{74: GETBP}
[ 4 -999 55 56 0 2 ]{75: CSTI 2}
[ 4 -999 55 56 0 2 2 ]{77: ADD}
[ 4 -999 55 56 0 4 ]{78: GETBP}
[ 4 -999 55 56 0 4 2 ]{79: SUB}
[ 4 -999 55 56 0 2 ]{80: GETBP}
[ 4 -999 55 56 0 2 2 ]{81: CSTI 2}
[ 4 -999 55 56 0 2 2 2 ]{83: ADD}
[ 4 -999 55 56 0 2 4 ]{84: LDI}
[ 4 -999 55 56 0 2 0 ]{85: ADD}
[ 4 -999 55 56 0 2 ]{86: LDI}
[ 4 -999 55 56 0 55 ]{87: PRINTI}
55 [ 4 -999 55 56 0 55 ]{88: INCSP -1}
[ 4 -999 55 56 0 ]{90: GETBP}
[ 4 -999 55 56 0 2 ]{91: CSTI 2}
[ 4 -999 55 56 0 2 2 ]{93: ADD}
[ 4 -999 55 56 0 4 ]{94: GETBP}
[ 4 -999 55 56 0 4 2 ]{95: CSTI 2}
[ 4 -999 55 56 0 4 2 2 ]{97: ADD}
[ 4 -999 55 56 0 4 4 ]{98: LDI}
```

```
[ 4 -999 55 56 0 4 0 ]{99: CSTI 1}
[ 4 -999 55 56 0 4 0 1 ]{101: ADD}
[ 4 -999 55 56 0 4 1 ]{102: STI}
[ 4 -999 55 56 1 1 ]{103: INCSP -1}
[ 4 -999 55 56 1 ]{105: INCSP 0}
[ 4 -999 55 56 1 ]{107: GETBP}
[ 4 -999 55 56 1 2 ]{108: CSTI 2}
[ 4 -999 55 56 1 2 2 ]{110: ADD}
[ 4 -999 55 56 1 4 ]{111: LDI}
[ 4 -999 55 56 1 1 ]{112: CSTI 2}
[ 4 -999 55 56 1 1 2 ]{114: LT}
[ 4 -999 55 56 1 1 ]{115: IFNZRO 74}
[ 4 -999 55 56 1 ]{74: GETBP}
[ 4 -999 55 56 1 2 ]{75: CSTI 2}
[ 4 -999 55 56 1 2 2 ]{77: ADD}
[ 4 -999 55 56 1 4 ]{78: GETBP}
[ 4 -999 55 56 1 4 2 ]{79: SUB}
[ 4 -999 55 56 1 2 ]{80: GETBP}
[ 4 -999 55 56 1 2 2 ]{81: CSTI 2}
[ 4 -999 55 56 1 2 2 2 ]{83: ADD}
[ 4 -999 55 56 1 2 4 ]{84: LDI}
[ 4 -999 55 56 1 2 1 ]{85: ADD}
[ 4 -999 55 56 1 3 ]{86: LDI}
[ 4 -999 55 56 1 56 ]{87: PRINTI}
56 [ 4 -999 55 56 1 56 ]{88: INCSP -1}
[ 4 -999 55 56 1 ]{90: GETBP}
[ 4 -999 55 56 1 2 ]{91: CSTI 2}
[ 4 -999 55 56 1 2 2 ]{93: ADD}
[ 4 -999 55 56 1 4 ]{94: GETBP}
[ 4 -999 55 56 1 4 2 ]{95: CSTI 2}
[ 4 -999 55 56 1 4 2 2 ]{97: ADD}
[ 4 -999 55 56 1 4 4 ]{98: LDI}
[ 4 -999 55 56 1 4 1 ]{99: CSTI 1}
[ 4 -999 55 56 1 4 1 1 ]{101: ADD}
[ 4 -999 55 56 1 4 2 ]{102: STI}
[ 4 -999 55 56 2 2 ]{103: INCSP -1}
[ 4 -999 55 56 2 ]{105: INCSP 0}
[ 4 -999 55 56 2 ]{107: GETBP}
[ 4 -999 55 56 2 2 ]{108: CSTI 2}
[ 4 -999 55 56 2 2 2 ]{110: ADD}
[ 4 -999 55 56 2 4 ]{111: LDI}
[ 4 -999 55 56 2 2 ]{112: CSTI 2}
[ 4 -999 55 56 2 2 2 ]{114: LT}
[ 4 -999 55 56 2 0 ]{115: IFNZRO 74}
[ 4 -999 55 56 2 ]{117: INCSP -3}
[ 4 -999 ]{119: RET -1}
[ -999 ]{4: STOP}

Ran 0.024 seconds
```

C. MicroML

C.1 Makefile for MicroML

```
.PHONY: fsi
fsi:
  fsyacc --module FunPar FunPar.fsy
  fslex --unicode FunLex.fsl
  fsharpi -r ~/fsharp/FsLexYacc.Runtime.dll Absyn.fs FunPar.fs FunLex.fs Parse.fs

.PHONY: fsi-higher
fsi-higher:
  fsyacc --module FunPar FunPar.fsy
  fslex --unicode FunLex.fsl
  fsharpi -r ~/fsharp/FsLexYacc.Runtime.dll Absyn.fs FunPar.fs FunLex.fs Parse.fs
    HigherFun.fs ParseAndRunHigher.fs
```

C.2 Afledsnignsregler for typebestemmelse for MicroML

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

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

Figur C.1: Afledningsregler for typebestemmelse af programmer med operatorene `@` og `=` i MicroML sproget. Kopieret fra opgavebeskrivelsen