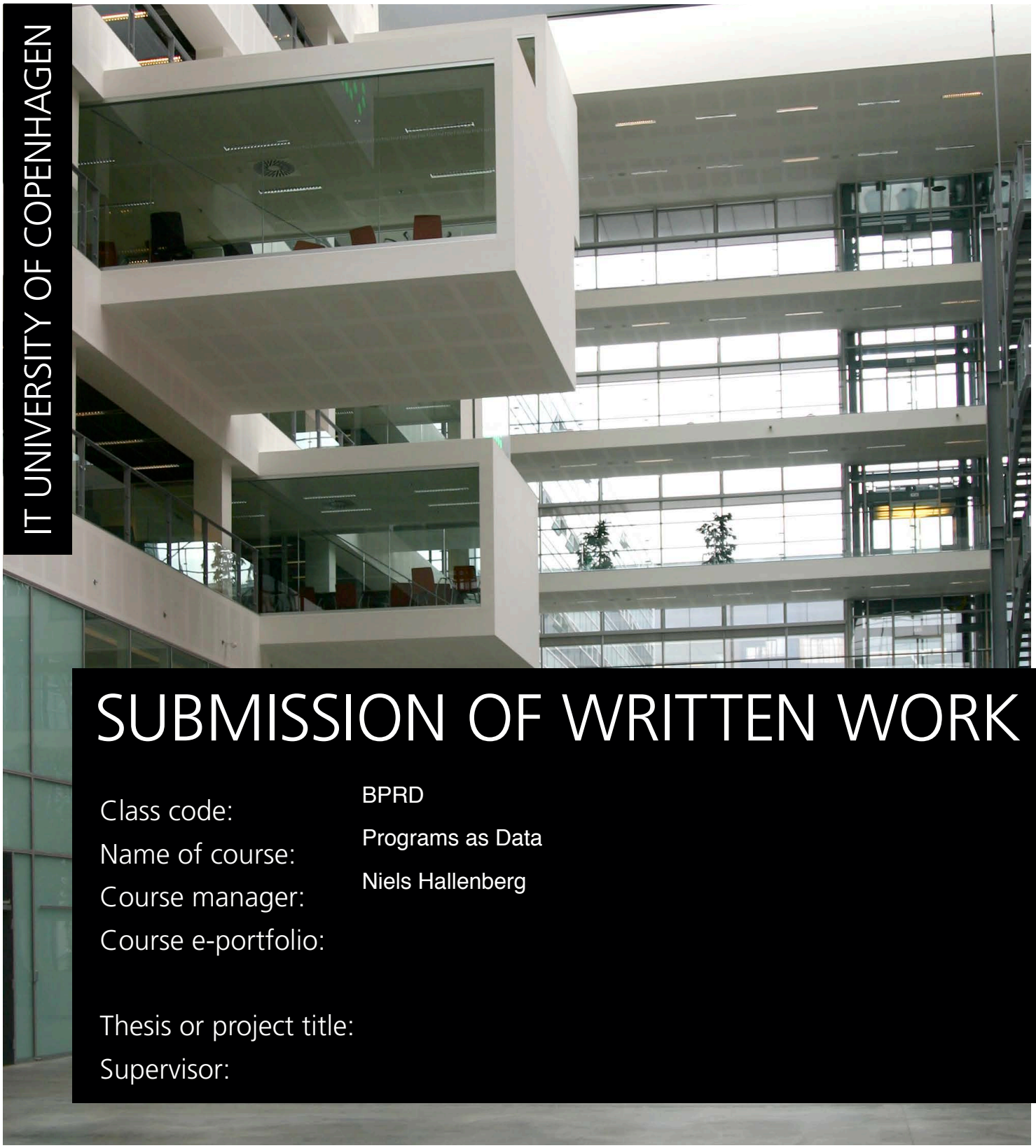
IT UNIVERSITY OF COPENHAGEN



# 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 13, 2015

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

Jacob Cholewa

*Signature*

13/12 2014

*Date*



Jacob Benjamin Cholewa jbec@itu.dk

IT-UNIVERSITETET I KØBENHAVN

# 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, $f(etch)$. 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:

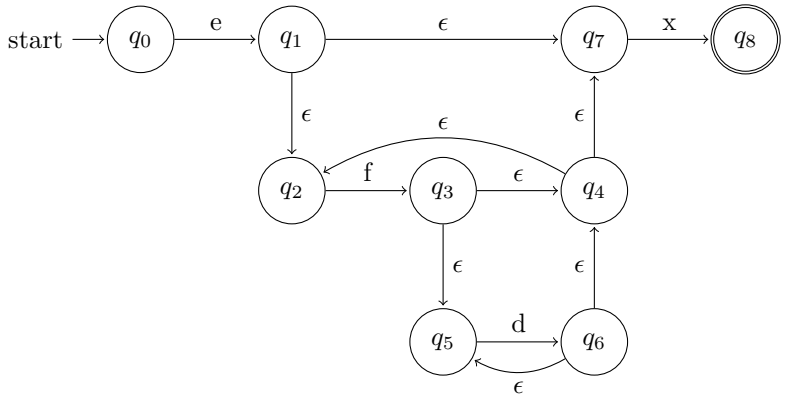
$$ex,\ efx,\ effx,\ efdx,\ efffdx,\ efddfddfx$$

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





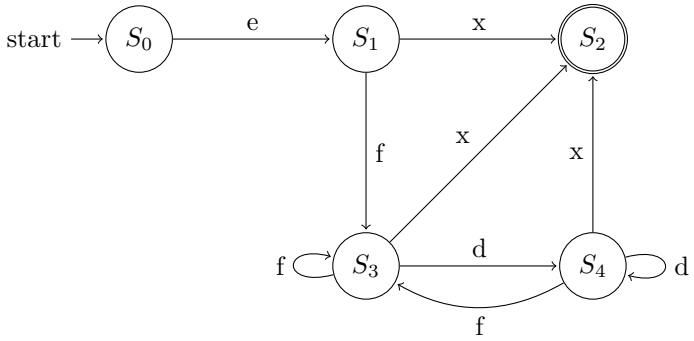
Jacob Benjamin Cholewa jbec@itu.dk

## 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 $\epsilon$ 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 $\epsilon$ 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:

$$f[sd]*$$

# 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

I denne opgave har jeg lavet fire udtryk til at teste min implementation af `Pair, Fst` og `Snd`. Jeg bruger til evalueringen et lille hjælpe program `run` som kan findes i appendiks A. 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 er at evaluere `Fst` og `Snd` ved at *Pattern Matche* på `Pair` og ikke på evalueringen af `Pair`. 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

I denne opgave har 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

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

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

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

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


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 $n = a - q$.

| | $q$ | $q+1$ | | $q+n-1$ | $a$ | |
|---|---|---|---|---|---|---|
| ... | $arr[0]$ | $arr[1]$ | ... | $arr[n-1]$ | $q$ | ... |

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

$$a = s[sp]$$
$$q = s[s[sp]]$$
$$n = a - q$$

Denne logik er implementeret i den abstrakte maskine.

```
final static int
  ...
  STOP = 25,
  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` og `exarrlen.c` 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. Programmet vist i 3.3 giver os altså længden mellem det første element i arrayet og $b$'s placering på stakken i stedet for arrayets rigtige længde.

| | $q$ | $q+1$ | | $q+n-1$ | $a$ | | $b$ | | $b*$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| ... | $arr[0]$ | $arr[1]$ | ... | $arr[n-1]$ | $q$ | ... | $q$ | ... | $a$ | ... |

Programmet vist i 3.4 virke da vi her laver en pointer $b*$ der før kaldet til `ARRLEN` bliver derefereret til $a$.

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

```
$ ./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 `DOTDOT`. 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 Kode 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 Kode til opgave 3 og 4

## 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

# C Fun kode

I dette appendiks ses alle de opgivede Fun filer jeg har ændret i

## FunLex.fsl

```
{
 (* File Fun/Funlex.fsl
    Lexer for a simple functional language (micro-ML)
    sestoft@itu.dk * 2010-01-02
  *)

module FunLex

open Microsoft.FSharp.Text.Lexing
open FunPar;

let lexemeAsString lexbuf =
    LexBuffer<char>.LexemeString lexbuf

(* Start of outermost comment currently being scanned *)
let commentStart = ref Position.Empty;

let commentDepth = ref 0; (* Current comment nesting *)

(* Distinguish keywords from identifiers: *)

let keyword s =
    match s with
    | "else" -> ELSE
    | "end"  -> END
    | "false" -> CSTBOOL false
    | "if"   -> IF
    | "in"   -> IN
    | "let"  -> LET
    | "not"  -> NOT
    | "then" -> THEN
    | "true" -> CSTBOOL true
    | "fst"  -> FST
    | "snd"  -> SND
    | _      -> NAME s
}

rule Token = parse
  | [' ' '\t' '\r'] { Token lexbuf }
  | '\n'          { lexbuf.EndPos <- lexbuf.EndPos.NextLine; Token lexbuf }
  | ['0'-'9']+    { CSTINT (System.Int32.Parse (lexemeAsString lexbuf)) }
  | ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
                  { keyword (lexemeAsString lexbuf) }
  | "(*"          { commentStart := lexbuf.StartPos;
                    commentDepth := 1;
                    SkipComment lexbuf; Token lexbuf }
  | '='           { EQ }
  | "<>"          { NE }
```

```
  | '>'            { GT }
  | '<'            { LT }
  | ">="           { GE }
  | "<="           { LE }
  | '+'            { PLUS }
  | '-'            { MINUS }
  | '*'            { TIMES }
  | '/'            { DIV }
  | '%'            { MOD }
  | '('            { LPAR }
  | ')'            { RPAR }
  | ','            { COMMA }
  | eof            { EOF }
  | _              { failwith "Lexer error: illegal symbol" }

and SkipComment = parse
    "*)"           { commentDepth := !commentDepth - 1;
                     if !commentDepth = 0 then ()
                     else SkipComment lexbuf
                   }
   | "(*"          { commentDepth := !commentDepth + 1;
                     SkipComment lexbuf }
   | eof           { failwith "Lexer error: unterminated comment" }
   | _             { SkipComment lexbuf }
```

# FunPar.fsy

```
%{
 (* File Fun/FunPar.fsy
    Parser for micro-ML, a small functional language; one-argument functions.
    sestoft@itu.dk * 2009-10-19
  *)

 open Absyn;
%}

%token <int> CSTINT
%token <string> NAME
%token <bool> CSTBOOL
%token COMMA FST SND
%token ELSE END FALSE IF IN LET NOT THEN TRUE
%token PLUS MINUS TIMES DIV MOD
%token EQ NE GT LT GE LE
%token LPAR RPAR
%token EOF

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

%start Main
%type <Absyn.expr> Main Expr AtExpr Const
%type <Absyn.expr> AppExpr
```

```
%%

Main:
    Expr EOF                          { $1 }
;

Expr:
    AtExpr                            { $1                 }
  | AppExpr                           { $1                 }
  | IF Expr THEN Expr ELSE Expr       { If($2, $4, $6)     }
  | LPAR Expr COMMA Expr RPAR         { Pair($2,$4)        }
  | FST Expr                          { Fst($2)            }
  | SND Expr                          { Snd($2)            }
  | Expr PLUS Expr                    { Prim("+", $1, $3)  }
  | Expr MINUS Expr                   { Prim("-", $1, $3)  }
  | Expr TIMES Expr                   { Prim("*", $1, $3)  }
  | Expr DIV  Expr                    { Prim("/", $1, $3)  }
  | Expr MOD  Expr                    { Prim("%", $1, $3)  }
  | Expr EQ   Expr                    { Prim("=", $1, $3)  }
  | Expr NE   Expr                    { Prim("<>", $1, $3) }
  | Expr GT   Expr                    { Prim(">", $1, $3)  }
  | Expr LT   Expr                    { Prim("<", $1, $3)  }
  | Expr GE   Expr                    { Prim(">=", $1, $3) }
  | Expr LE   Expr                    { Prim("<=", $1, $3) }
;

AtExpr:
    Const                             { $1                 }
  | NAME                              { Var $1             }
  | LET NAME EQ Expr IN Expr END      { Let($2, $4, $6)    }
  | LET NAME NAME EQ Expr IN Expr END { Letfun($2, $3, $5, $7) }
  | LPAR Expr RPAR                    { $2                 }
;

AppExpr:
    AtExpr AtExpr                     { Call($1, $2)       }
  | AppExpr AtExpr                    { Call($1, $2)       }
;

Const:
    CSTINT                            { CstI($1)           }
  | CSTBOOL                           { CstB($1)           }
;
```

# Absyn.fs

```
(* Fun/Absyn.fs * Abstract syntax for micro-ML, a functional language *)

module Absyn

type expr =
  | Pair of expr * expr
  | Fst of expr
  | Snd of expr
  | CstI of int
  | CstB of bool
  | Var of string
```

```
  | Let of string * expr * expr
  | Prim of string * expr * expr
  | If of expr * expr * expr
  | Letfun of string * string * expr * expr (* (f, x, fBody, letBody) *)
  | Call of expr * expr
```

## HigherFun.fs

```
(* A functional language with integers and higher-order functions
   sestoft@itu.dk 2009-09-11

   The language is higher-order because the value of an expression may
   be a function (and therefore a function can be passed as argument
   to another function).

   A function definition can have only one parameter, but a
   multiparameter (curried) function can be defined using nested
   function definitions:

      let f x = let g y = x + y in g end in f 6 7 end
 *)

module HigherFun

open Absyn

(* Environment operations *)

type 'v env = (string * 'v) list

let rec lookup env x =
    match env with
    | []       -> failwith (x + " not found")
    | (y, v)::r -> if x=y then v else lookup r x;;

(* A runtime value is an integer or a function closure *)

type value =
  | Int of int
  | ValuePair of value * value
  | Closure of string * string * expr * value env  (* (f, x, fBody, fDeclEnv) *)

let rec eval (e : expr) (env : value env) : value =
    match e with
    | CstI i -> Int i
    | CstB b -> Int (if b then 1 else 0)
    | Var x -> lookup env x
    | Pair(e1,e2)   -> ValuePair((eval e1 env),(eval e2 env))
    | Fst(Pair(e1,e2)) -> eval e1 env
    | Snd(Pair(e1,e2)) -> eval e2 env
    | Prim(ope, e1, e2) ->
      let v1 = eval e1 env
      let v2 = eval e2 env
      match (ope, v1, v2) with
      | ("*", Int i1, Int i2) -> Int (i1 * i2)
      | ("+", Int i1, Int i2) -> Int (i1 + i2)
      | ("-", Int i1, Int i2) -> Int (i1 - i2)
```

```
      | ("=", Int i1, Int i2) -> Int (if i1 = i2 then 1 else 0)
      | ("<", Int i1, Int i2) -> Int (if i1 < i2 then 1 else 0)
      |  _ -> failwith "unknown primitive or wrong type"
    | Let(x, eRhs, letBody) ->
      let xVal = eval eRhs env
      let letEnv = (x, xVal) :: env
      eval letBody letEnv
    | If(e1, e2, e3) ->
      match eval e1 env with
      | Int 0 -> eval e3 env
      | Int _ -> eval e2 env
      | _     -> failwith "eval If"
    | Letfun(f, x, fBody, letBody) ->
      let bodyEnv = (f, Closure(f, x, fBody, env)) :: env
      eval letBody bodyEnv
    | Call(eFun, eArg) ->
      let fClosure = eval eFun env
      match fClosure with
      | Closure (f, x, fBody, fDeclEnv) ->
        let xVal = eval eArg env
        let fBodyEnv = (x, xVal) :: (f, fClosure) :: fDeclEnv
        in eval fBody fBodyEnv
      | _ -> failwith "eval Call: not a function"
    | _               -> failwith "Invalid syntax"

(* Evaluate in empty environment: program must have no free variables: *)

let run e = eval e [];;

(* Examples in abstract syntax *)

let ex1 = Letfun("f1", "x", Prim("+", Var "x", CstI 1),
                Call(Var "f1", CstI 12));

(* Factorial *)

let ex2 = Letfun("fac", "x",
                If(Prim("=", Var "x", CstI 0),
                   CstI 1,
                   Prim("*", Var "x",
                            Call(Var "fac",
                                 Prim("-", Var "x", CstI 1)))),
                Call(Var "fac", Var "n"));

(* let fac10 = eval ex2 [("n", Int 10)];; *)

let ex3 =
    Letfun("tw", "g",
           Letfun("app", "x", Call(Var "g", Call(Var "g", Var "x")),
                  Var "app"),
           Letfun("mul3", "y", Prim("*", CstI 3, Var "y"),
                  Call(Call(Var "tw", Var "mul3"), CstI 11)));;

let ex4 =
    Letfun("tw", "g",
           Letfun("app", "x", Call(Var "g", Call(Var "g", Var "x")),
                  Var "app"),
           Letfun("mul3", "y", Prim("*", CstI 3, Var "y"),
```

```
                Call(Var "tw", Var "mul3")));;

(* let add1 x = x + 1 in add1 end *)
let add1 = Letfun("add1", "x",
                Prim("+", Var "x", CstI 1),
                Var "add1")
let add1C = eval add1 []
let add1with2 = eval (Call(Var "add1C", CstI 2)) [("add1C",add1C)]

(* let tw g = let app y = g (g y) in app end in tw end *)
let tw =
  Letfun("tw", "g",
    Letfun("app", "y", Call(Var "g", Call(Var "g", Var "y")), Var "app"),
    Var "tw")
let twC = eval tw []


let twAdd1C = eval (Call(Var "tw", Var "add1")) [("tw",twC);("add1",add1C)]
let res = eval (Call(Var "twAdd1C", CstI 1)) [("twAdd1C",twAdd1C)]
```

# D MicroC kode

I dette appendiks ses alle de opgivede Micro filer jeg har ændret i

## CLex.fsl

```
{
 (* File MicroC/CLex.lex
    Lexer specification for micro-C, a small imperative language
  *)

module CLex

open Microsoft.FSharp.Text.Lexing
open CPar;

let lexemeAsString lexbuf =
    LexBuffer<char>.LexemeString lexbuf

(* Scan keywords as identifiers and use this function to distinguish them. *)
(* If the set of keywords is large, use a hashtable instead.       *)

let keyword s =
    match s with
    | "char"   -> CHAR
    | "else"   -> ELSE
    | "false"  -> CSTBOOL 0
    | "if"     -> IF
    | "int"    -> INT
    | "null"   -> NULL
    | "print"  -> PRINT
    | "println" -> PRINTLN
    | "return" -> RETURN
    | "true"   -> CSTBOOL 1
    | "void"   -> VOID
    | "while"  -> WHILE
    | _        -> NAME s

let cEscape s =
    match s with
    | "\\\\" -> '\\'
    | "\\\"" -> '\"'
    | "\\a" -> '\007'
    | "\\b" -> '\008'
    | "\\t" -> '\t'
    | "\\n" -> '\n'
    | "\\v" -> '\011'
    | "\\f" -> '\012'
    | "\\r" -> '\r'
    | _     -> failwith "Lexer error: impossible C escape"
}

rule Token = parse
  | [' ' '\t' '\r'] { Token lexbuf }
```

```
  | '\n'          { lexbuf.EndPos <- lexbuf.EndPos.NextLine; Token lexbuf }
  | ['0'-'9']+    { CSTINT (System.Int32.Parse (lexemeAsString lexbuf)) }
  | ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
                  { keyword (lexemeAsString lexbuf) }
  | '+'           { PLUS }
  | '-'           { MINUS }
  | '*'           { TIMES }
  | '/'           { DIV }
  | '%'           { MOD }
  | '='           { ASSIGN }
  | "=="          { EQ }
  | "!="          { NE }
  | '>'           { GT }
  | '<'           { LT }
  | ">="          { GE }
  | "<="          { LE }
  | "||"          { SEQOR }
  | "&&"          { SEQAND }
  | "&"           { AMP }
  | "!"           { NOT }
  | '('           { LPAR }
  | ')'           { RPAR }
  | '{'           { LBRACE }
  | '}'           { RBRACE }
  | '['           { LBRACK }
  | ']'           { RBRACK }
  | ';'           { SEMI }
  | ','           { COMMA }
  | ".."          { DOTDOT }
  | '|'           { BAR }
  | "//"          { EndLineComment lexbuf; Token lexbuf }
  | "/*"          { Comment lexbuf; Token lexbuf }
  | '"'           { CSTSTRING (String [] lexbuf) }
  | eof           { EOF }
  | _             { failwith "Lexer error: illegal symbol" }

and Comment = parse
   | "/*"         { Comment lexbuf; Comment lexbuf }
   | "*/"         { () }
   | '\n'         { lexbuf.EndPos <- lexbuf.EndPos.NextLine; Comment lexbuf }
   | (eof | '\026') { failwith "Lexer error: unterminated comment" }
   | _            { Comment lexbuf }

and EndLineComment = parse
   | '\n'         { lexbuf.EndPos <- lexbuf.EndPos.NextLine }
   | (eof | '\026') { () }
   | _            { EndLineComment lexbuf }

and String chars = parse
  | '"'
     { Microsoft.FSharp.Core.String.concat "" (List.map string (List.rev chars)) }
  | '\\' ['\\' '"' 'a' 'b' 't' 'n' 'v' 'f' 'r']
     { String (cEscape (lexemeAsString lexbuf) :: chars) lexbuf }
  | "''"
     { String ('\'' :: chars) lexbuf }
  | '\\'
     { failwith "Lexer error: illegal escape sequence" }
  | (eof | '\026')
```

```
    { failwith "Lexer error: unterminated string" }
  | ['\n' '\r']
    { failwith "Lexer error: newline in string" }
  | ['\000'-'\031' '\127' '\255']
    { failwith "Lexer error: invalid character in string" }
  | _
    { String (char (lexbuf.LexemeChar 0) :: chars) lexbuf }
```

# CPar.fsy

```
%{
(*      File MicroC/CPar.fsy
        Parser specification for micro-C, a small imperative language
        sestoft@itu.dk * 2009-09-29
        No (real) shift/reduce conflicts thanks to Niels Kokholm.
*)

open Absyn

let compose1 f (g, s) = ((fun x -> g(f(x))), s)
let nl = CstI 10
%}

%token <int> CSTINT CSTBOOL
%token <string> CSTSTRING NAME

%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 BAR DOTDOT
%token NOT SEQOR SEQAND
%token LPAR RPAR LBRACE RBRACE LBRACK RBRACK SEMI COMMA ASSIGN AMP
%token EOF

%right ASSIGN           /* lowest precedence */
%nonassoc PRINT
%left SEQOR
%left SEQAND
%left EQ NE
%nonassoc GT LT GE LE
%left PLUS MINUS
%left TIMES DIV MOD
%nonassoc NOT AMP
%nonassoc LBRACK        /* highest precedence */

%start Main
%type <Absyn.program> Main

%%

Main:
    Topdecs EOF                     { Prog $1 }
;

Topdecs:
    /* empty */                     { [] }
  | Topdec Topdecs                  { $1 :: $2 }
```

```
;

Topdec:
    Vardec SEMI                     { Vardec (fst $1, snd $1) }
  | Fundec                          { $1 }
;

Vardec:
    Type Vardesc                    { ((fst $2) $1, snd $2) }
;

Vardesc:
    NAME                            { ((fun t -> t), $1)                }
  | TIMES Vardesc                   { compose1 TypP $2                  }
  | LPAR Vardesc RPAR               { $2                                }
  | Vardesc LBRACK RBRACK           { compose1 (fun t -> TypA(t, None)) $1 }
  | Vardesc LBRACK CSTINT RBRACK    { compose1 (fun t -> TypA(t, Some $3)) $1 }
  | Vardesc LBRACK Number DOTDOT Number DOTDOT Number RBRACK
                                    { compose1 (fun t -> TypR(t,$3,$5,$7)) $1 }
;

Fundec:
    VOID NAME LPAR Paramdecs RPAR Block { Fundec(None, $2, $4, $6) }
  | Type NAME LPAR Paramdecs RPAR Block { Fundec(Some($1), $2, $4, $6) }
;

Paramdecs:
    /* empty */                     { [] }
  | Paramdecs1                      { $1 }
;

Paramdecs1:
    Vardec                          { [$1]    }
  | Vardec COMMA Paramdecs1         { $1 :: $3 }
;

Block:
    LBRACE StmtOrDecSeq RBRACE      { Block $2 }
;

StmtOrDecSeq:
    /* empty */                     { [] }
  | Stmt StmtOrDecSeq               { Stmt $1 :: $2 }
  | Vardec SEMI StmtOrDecSeq        { Dec (fst $1, snd $1) :: $3 }
;

Stmt:
    StmtM                           { $1 }
  | StmtU                           { $1 }
;

StmtM: /* No unbalanced if-else */
    Expr SEMI                       { Expr($1)          }
  | RETURN SEMI                     { Return None       }
  | RETURN Expr SEMI                { Return(Some($2))  }
  | Block                           { $1                }
  | IF LPAR Expr RPAR StmtM ELSE StmtM { If($3, $5, $7) }
  | WHILE LPAR Expr RPAR StmtM      { While($3, $5)     }
```

```
;

StmtU:
    IF LPAR Expr RPAR StmtM ELSE StmtU { If($3, $5, $7)  }
  | IF LPAR Expr RPAR Stmt          { If($3, $5, Block []) }
  | WHILE LPAR Expr RPAR StmtU      { While($3, $5)      }
;

Expr:
    Access                          { Access $1         }
  | ExprNotAccess                   { $1                }
;

ExprNotAccess:
    AtExprNotAccess                 { $1                }
  | Access ASSIGN Expr              { Assign($1, $3)    }
  | NAME LPAR Exprs RPAR            { Call($1, $3)      }
  | NOT Expr                        { Prim1("!", $2)    }
  | PRINT Expr                      { Prim1("printi", $2) }
  | PRINTLN                         { Prim1("printc", nl) }
  | BAR Expr BAR                    { Prim1("arrlen", $2) }
  | Expr PLUS Expr                  { Prim2("+", $1, $3) }
  | Expr MINUS Expr                 { Prim2("-", $1, $3) }
  | Expr TIMES Expr                 { Prim2("*", $1, $3) }
  | Expr DIV  Expr                  { Prim2("/", $1, $3) }
  | Expr MOD  Expr                  { Prim2("%", $1, $3) }
  | Expr EQ   Expr                  { Prim2("==", $1, $3) }
  | Expr NE   Expr                  { Prim2("!=", $1, $3) }
  | Expr GT   Expr                  { Prim2(">", $1, $3) }
  | Expr LT   Expr                  { Prim2("<", $1, $3) }
  | Expr GE   Expr                  { Prim2(">=", $1, $3) }
  | Expr LE   Expr                  { Prim2("<=", $1, $3) }
  | Expr SEQAND Expr                { Andalso($1, $3)   }
  | Expr SEQOR Expr                 { Orelse($1, $3)    }
;

AtExprNotAccess:
    Const                           { CstI $1           }
  | LPAR ExprNotAccess RPAR         { $2                }
  | AMP Access                      { Addr $2           }
;

Access:
    NAME                            { AccVar $1         }
  | LPAR Access RPAR                { $2                }
  | TIMES Access                    { AccDeref (Access $2)}
  | TIMES AtExprNotAccess           { AccDeref $2       }
  | Access LBRACK Expr RBRACK       { AccIndex($1, $3)  }
;

Exprs:
    /* empty */                     { []      }
  | Exprs1                          { $1      }
;

Exprs1:
    Expr                            { [$1]    }
  | Expr COMMA Exprs1               { $1 :: $3 }
```

```
;

Const:
    Number                          { $1      }
  | CSTBOOL                         { $1      }
  | NULL                            { -1      }
;

Number:
    CSTINT                          { $1      }
  | MINUS CSTINT                    { - $2    }
;

Type:
    INT                             { TypI    }
  | CHAR                            { TypC    }
;
```

# Absyn.fs

```
(* File MicroC/Absyn.fs
   Abstract syntax of micro-C, an imperative language.
   sestoft@itu.dk 2009-09-25

   Must precede Interp.fs, Comp.fs and Contcomp.fs in Solution Explorer
 *)

module Absyn

type typ =
  | TypI                          (* Type int                *)
  | TypC                          (* Type char               *)
  | TypA of typ * int option      (* Array type              *)
  | TypP of typ                   (* Pointer type            *)
  | TypR of typ * int * int * int (* Array range             *)

and expr =
  | Access of access              (* x    or  *p    or  a[e]    *)
  | Assign of access * expr       (* x=e  or  *p=e  or  a[e]=e  *)
  | Addr of access                (* &x   or  &*p   or  &a[e]   *)
  | CstI of int                   (* Constant                *)
  | Prim1 of string * expr        (* Unary primitive operator *)
  | Prim2 of string * expr * expr (* Binary primitive operator *)
  | Andalso of expr * expr        (* Sequential and          *)
  | Orelse of expr * expr         (* Sequential or           *)
  | Call of string * expr list    (* Function call f(...)    *)

and access =
  | AccVar of string              (* Variable access     x   *)
  | AccDeref of expr              (* Pointer dereferencing *p *)
  | AccIndex of access * expr     (* Array indexing      a[e] *)

and stmt =
  | If of expr * stmt * stmt      (* Conditional             *)
  | While of expr * stmt          (* While loop              *)
  | Expr of expr                  (* Expression statement e;  *)
  | Return of expr option         (* Return from method      *)
```

```
  | Block of stmtordec list       (* Block: grouping and scope *)

and stmtordec =
  | Dec of typ * string           (* Local variable declaration *)
  | Stmt of stmt                  (* A statement                *)

and topdec =
  | Fundec of typ option * string * (typ * string) list * stmt
  | Vardec of typ * string

and program =
  | Prog of topdec list
```

# Comp.fs

```
(* File MicroC/Comp.fs
   A compiler from micro-C, a sublanguage of the C language, to an
   abstract machine. Direct (forwards) compilation without
   optimization of jumps to jumps, tail-calls etc.
   sestoft@itu.dk * 2009-09-23, 2011-11-10

   A value is an integer; it may represent an integer or a pointer,
   where a pointer is just an address in the store (of a variable or
   pointer or the base address of an array).

   The compile-time environment maps a global variable to a fixed
   store address, and maps a local variable to an offset into the
   current stack frame, relative to its bottom. The run-time store
   maps a location to an integer. This freely permits pointer
   arithmetics, as in real C. A compile-time function environment
   maps a function name to a code label. In the generated code,
   labels are replaced by absolute code addresses.

   Expressions can have side effects. A function takes a list of
   typed arguments and may optionally return a result.

   Arrays can be one-dimensional and constant-size only. For
   simplicity, we represent an array as a variable which holds the
   address of the first array element. This is consistent with the
   way array-type parameters are handled in C, but not with the way
   array-type variables are handled. Actually, this was how B (the
   predecessor of C) represented array variables.

   The store behaves as a stack, so all data except global variables
   are stack allocated: variables, function parameters and arrays.
*)

module Comp

open System.IO
open Absyn
open Machine

(* ------------------------------------------------------------------- *)

(* Simple environment operations *)
```

```
type 'data env = (string * 'data) list

let rec lookup env x =
    match env with
    | []        -> failwith (x + " not found")
    | (y, v)::yr -> if x=y then v else lookup yr x

(* A global variable has an absolute address, a local one has an offset: *)

type var =
     | Glovar of int                (* absolute address in stack       *)
     | Locvar of int                (* address relative to bottom of frame *)

(* The variable environment keeps track of global and local variables, and
   keeps track of next available offset for local variables *)

type varEnv = (var * typ) env * int

(* The function environment maps function name to label and parameter decs *)

type paramdecs = (typ * string) list
type funEnv = (label * typ option * paramdecs) env

(* Bind declared variable in env and generate code to allocate it: *)

let allocate (kind : int -> var) (typ, x) (varEnv : varEnv) : varEnv * instr list =
    let (env, fdepth) = varEnv
    match typ with
    | TypA (TypA _, _) ->
      raise (Failure "allocate: array of arrays not permitted")
    | TypA (t, Some i) ->
      let newEnv = ((x, (kind (fdepth+i), typ)) :: env, fdepth+i+1)
      let code = [INCSP i; GETSP; CSTI (i-1); SUB]
      (newEnv, code)
    | 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"
    | _ ->
      let newEnv = ((x, (kind (fdepth), typ)) :: env, fdepth+1)
      let code = [INCSP 1]
      (newEnv, code)

(* Bind declared parameters in env: *)

let bindParam (env, fdepth) (typ, x) : varEnv =
    ((x, (Locvar fdepth, typ)) :: env , fdepth+1)

let bindParams paras ((env, fdepth) : varEnv) : varEnv =
    List.fold bindParam (env, fdepth) paras;

(* ----------------------------------------------------------------- *)
```

```
(* Build environments for global variables and functions *)

let makeGlobalEnvs (topdecs : topdec list) : varEnv * funEnv * instr list =
    let rec addv decs varEnv funEnv =
        match decs with
        | []         -> (varEnv, funEnv, [])
        | dec::decr ->
          match dec with
          | Vardec (typ, var) ->
            let (varEnv1, code1)     = allocate Glovar (typ, var) varEnv
            let (varEnvr, funEnvr, coder) = addv decr varEnv1 funEnv
            (varEnvr, funEnvr, code1 @ coder)
          | Fundec (tyOpt, f, xs, body) ->
            addv decr varEnv ((f, (newLabel(), tyOpt, xs)) :: funEnv)
    addv topdecs ([], 0) []

(* ----------------------------------------------------------------------- *)

(* Compiling micro-C statements:
   * stmt   is the statement to compile
   * varenv is the local and global variable environment
   * funEnv is the global function environment
*)

let rec cStmt stmt (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match stmt with
    | If(e, stmt1, stmt2) ->
      let labelse = newLabel()
      let labend = newLabel()
      cExpr e varEnv funEnv @ [IFZERO labelse]
      @ cStmt stmt1 varEnv funEnv @ [GOTO labend]
      @ [Label labelse] @ cStmt stmt2 varEnv funEnv
      @ [Label labend]
    | While(e, body) ->
      let labbegin = newLabel()
      let labtest = newLabel()
      [GOTO labtest; Label labbegin] @ cStmt body varEnv funEnv
      @ [Label labtest] @ cExpr e varEnv funEnv @ [IFNZRO labbegin]
    | Expr e ->
      cExpr e varEnv funEnv @ [INCSP -1]
    | Block stmts ->
      let rec loop stmts varEnv =
          match stmts with
          | []     -> (snd varEnv, [])
          | s1::sr ->
            let (varEnv1, code1) = cStmtOrDec s1 varEnv funEnv
            let (fdepthr, coder) = loop sr varEnv1
            (fdepthr, code1 @ coder)
      let (fdepthend, code) = loop stmts varEnv
      code @ [INCSP(snd varEnv - fdepthend)]
    | Return None ->
      [RET (snd varEnv - 1)]
    | Return (Some e) ->
      cExpr e varEnv funEnv @ [RET (snd varEnv)]

and cStmtOrDec stmtOrDec (varEnv : varEnv) (funEnv : funEnv) : varEnv * instr list =
    match stmtOrDec with
```

```
    | Stmt stmt   -> (varEnv, cStmt stmt varEnv funEnv)
    | Dec (typ, x) -> allocate Locvar (typ, x) varEnv

(* Compiling micro-C expressions:

   * e      is the expression to compile
   * varEnv is the local and gloval variable environment
   * funEnv is the global function environment

   Net effect principle: if the compilation (cExpr e varEnv funEnv) of
   expression e returns the instruction sequence instrs, then the
   execution of instrs will leave the rvalue of expression e on the
   stack top (and thus extend the current stack frame with one element).
*)

and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match e with
    | Access acc    -> cAccess acc varEnv funEnv @ [LDI]
    | Assign(acc, e) -> cAccess acc varEnv funEnv @ cExpr e varEnv funEnv @ [STI]
    | CstI i        -> [CSTI i]
    | Addr acc      -> cAccess acc varEnv funEnv
    | Prim1(ope, Access acc) when ope = "arrlen" -> cExpr (Addr acc) varEnv funEnv @
        [ARRLEN]
    | Prim1(ope, e1) ->
      cExpr e1 varEnv funEnv
      @ (match ope with
         | "!"      -> [NOT]
         | "printi" -> [PRINTI]
         | "printc" -> [PRINTC]
         | _        -> raise (Failure "unknown primitive 1"))
    | Prim2(ope, e1, e2) ->
      cExpr e1 varEnv funEnv
      @ cExpr e2 varEnv funEnv
      @ (match ope with
         | "*"  -> [MUL]
         | "+"  -> [ADD]
         | "-"  -> [SUB]
         | "/"  -> [DIV]
         | "%"  -> [MOD]
         | "==" -> [EQ]
         | "!=" -> [EQ; NOT]
         | "<"  -> [LT]
         | ">=" -> [LT; NOT]
         | ">"  -> [SWAP; LT]
         | "<=" -> [SWAP; LT; NOT]
         | _    -> raise (Failure "unknown primitive 2"))
    | Andalso(e1, e2) ->
      let labend  = newLabel()
      let labfalse = newLabel()
      cExpr e1 varEnv funEnv
      @ [IFZERO labfalse]
      @ cExpr e2 varEnv funEnv
      @ [GOTO labend; Label labfalse; CSTI 0; Label labend]
    | Orelse(e1, e2) ->
      let labend = newLabel()
      let labtrue = newLabel()
      cExpr e1 varEnv funEnv
      @ [IFNZRO labtrue]
```

```
      @ cExpr e2 varEnv funEnv
      @ [GOTO labend; Label labtrue; CSTI 1; Label labend]
    | Call(f, es) -> callfun f es varEnv funEnv

(* Generate code to access variable, dereference pointer or index array.
   The effect of the compiled code is to leave an lvalue on the stack. *)

and cAccess access varEnv funEnv : instr list =
    match access with
    | AccVar x ->
      match lookup (fst varEnv) x with
      | Glovar addr, _ -> [CSTI addr]
      | Locvar addr, _ -> [GETBP; CSTI addr; ADD]
    | AccDeref e -> cExpr e varEnv funEnv
    | AccIndex(acc, idx) -> cAccess acc varEnv funEnv
                            @ [LDI] @ cExpr idx varEnv funEnv @ [ADD]

(* Generate code to evaluate a list es of expressions: *)

and cExprs es varEnv funEnv : instr list =
    List.concat(List.map (fun e -> cExpr e varEnv funEnv) es)

(* Generate code to evaluate arguments es and then call function f: *)

and callfun f es varEnv funEnv : instr list =
    let (labf, tyOpt, paramdecs) = lookup funEnv f
    let argc = List.length es
    if argc = List.length paramdecs then
      cExprs es varEnv funEnv @ [CALL(argc, labf)]
    else
      raise (Failure (f + ": parameter/argument mismatch"))


(* Compile a complete micro-C program: globals, call to main, functions *)

let cProgram (Prog topdecs) : instr list =
    let _ = resetLabels ()
    let ((globalVarEnv, _), funEnv, globalInit) = makeGlobalEnvs topdecs
    let compilefun (tyOpt, f, xs, body) =
        let (labf, _, paras) = lookup funEnv f
        let (envf, fdepthf) = bindParams paras (globalVarEnv, 0)
        let code = cStmt body (envf, fdepthf) funEnv
        [Label labf] @ code @ [RET (List.length paras-1)]
    let functions =
        List.choose (function
                        | Fundec (rTy, name, argTy, body)
                                 -> Some (compilefun (rTy, name, argTy, body))
                        | Vardec _ -> None)
                    topdecs
    let (mainlab, _, mainparams) = lookup funEnv "main"
    let argc = List.length mainparams
    globalInit
    @ [LDARGS; CALL(argc, mainlab); STOP]
    @ List.concat functions

(* Compile a complete micro-C and write the resulting instruction list
   to file fname; also, return the program as a list of instructions.
 *)
```

```
let intsToFile (inss : int list) (fname : string) =
    File.WriteAllText(fname, String.concat " " (List.map string inss))

let compileToFile program fname =
    let instrs  = cProgram program
    let bytecode = code2ints instrs
    intsToFile bytecode fname
    instrs

(* Example programs are found in the files ex1.c, ex2.c, etc *)
```

# Machine.fs

```
(* File MicroC/Machine.fs

   Instructions and code emission for a stack-based
   abstract machine * sestoft@itu.dk 2009-09-23

   Implementations of the machine are found in file MicroC/Machine.java
   and MicroC/machine.c.

   Must precede Comp.fs and Contcomp.fs in the VS Solution Explorer.
 *)

module Machine

type label = string

type instr =
  | Label of label                (* symbolic label; pseudo-instruc. *)
  | CSTI of int                   (* constant                        *)
  | ADD                           (* addition                        *)
  | SUB                           (* subtraction                     *)
  | MUL                           (* multiplication                  *)
  | DIV                           (* division                        *)
  | MOD                           (* modulus                         *)
  | EQ                            (* equality: s[sp-1] == s[sp]      *)
  | LT                            (* less than: s[sp-1] < s[sp]      *)
  | NOT                           (* logical negation: s[sp] != 0    *)
  | DUP                           (* duplicate stack top             *)
  | SWAP                          (* swap s[sp-1] and s[sp]          *)
  | LDI                           (* get s[s[sp]]                    *)
  | STI                           (* set s[s[sp-1]]                  *)
  | GETBP                         (* get bp                          *)
  | GETSP                         (* get sp                          *)
  | INCSP of int                  (* increase stack top by m         *)
  | GOTO of label                 (* go to label                     *)
  | IFZERO of label               (* go to label if s[sp] == 0       *)
  | IFNZRO of label               (* go to label if s[sp] != 0       *)
  | CALL of int * label           (* move m args up 1, push pc, jump *)
  | TCALL of int * int * label    (* move m args down n, jump        *)
  | RET of int                    (* pop m and return to s[sp]       *)
  | PRINTI                        (* print s[sp] as integer          *)
  | PRINTC                        (* print s[sp] as character        *)
  | LDARGS                        (* load command line args on stack *)
  | STOP                          (* halt the abstract machine       *)
```

```
  | ARRLEN                          (* get s[sp] = s[sp] - s[s[sp]] *)

(* Generate new distinct labels *)

let (resetLabels, newLabel) =
    let lastlab = ref -1
    ((fun () -> lastlab := 0), (fun () -> (lastlab := 1 + !lastlab; "L" +
        (!lastlab).ToString())))

(* Simple environment operations *)

type 'data env = (string * 'data) list

let rec lookup env x =
    match env with
    | []        -> failwith (x + " not found")
    | (y, v)::yr -> if x=y then v else lookup yr x

(* An instruction list is emitted in two phases:
   * pass 1 builds an environment labenv mapping labels to addresses
   * pass 2 emits the code to file, using the environment labenv to
     resolve labels
 *)

(* These numeric instruction codes must agree with Machine.java: *)

let CODECSTI  = 0
let CODEADD   = 1
let CODESUB   = 2
let CODEMUL   = 3
let CODEDIV   = 4
let CODEMOD   = 5
let CODEEQ    = 6
let CODELT    = 7
let CODENOT   = 8
let CODEDUP   = 9
let CODESWAP  = 10
let CODELDI   = 11
let CODESTI   = 12
let CODEGETBP = 13
let CODEGETSP = 14
let CODEINCSP = 15
let CODEGOTO  = 16
let CODEIFZERO = 17
let CODEIFNZRO = 18
let CODECALL  = 19
let CODETCALL = 20
let CODERET   = 21
let CODEPRINTI = 22
let CODEPRINTC = 23
let CODELDARGS = 24
let CODESTOP  = 25
let CODEARRLEN = 26;

(* Bytecode emission, first pass: build environment that maps
   each label to an integer address in the bytecode.
 *)
```

```
let makelabenv (addr, labenv) instr =
    match instr with
    | Label lab     -> (addr, (lab, addr) :: labenv)
    | CSTI i        -> (addr+2, labenv)
    | ADD           -> (addr+1, labenv)
    | SUB           -> (addr+1, labenv)
    | MUL           -> (addr+1, labenv)
    | DIV           -> (addr+1, labenv)
    | MOD           -> (addr+1, labenv)
    | EQ            -> (addr+1, labenv)
    | LT            -> (addr+1, labenv)
    | NOT           -> (addr+1, labenv)
    | DUP           -> (addr+1, labenv)
    | SWAP          -> (addr+1, labenv)
    | LDI           -> (addr+1, labenv)
    | STI           -> (addr+1, labenv)
    | GETBP         -> (addr+1, labenv)
    | GETSP         -> (addr+1, labenv)
    | INCSP m       -> (addr+2, labenv)
    | GOTO lab      -> (addr+2, labenv)
    | IFZERO lab    -> (addr+2, labenv)
    | IFNZRO lab    -> (addr+2, labenv)
    | CALL(m,lab)   -> (addr+3, labenv)
    | TCALL(m,n,lab) -> (addr+4, labenv)
    | RET m         -> (addr+2, labenv)
    | PRINTI        -> (addr+1, labenv)
    | PRINTC        -> (addr+1, labenv)
    | LDARGS        -> (addr+1, labenv)
    | STOP          -> (addr+1, labenv)
    | ARRLEN        -> (addr+1, labenv)

(* Bytecode emission, second pass: output bytecode as integers *)

let rec emitints getlab instr ints =
    match instr with
    | Label lab     -> ints
    | CSTI i        -> CODECSTI  :: i :: ints
    | ADD           -> CODEADD   :: ints
    | SUB           -> CODESUB   :: ints
    | MUL           -> CODEMUL   :: ints
    | DIV           -> CODEDIV   :: ints
    | MOD           -> CODEMOD   :: ints
    | EQ            -> CODEEQ    :: ints
    | LT            -> CODELT    :: ints
    | NOT           -> CODENOT   :: ints
    | DUP           -> CODEDUP   :: ints
    | SWAP          -> CODESWAP  :: ints
    | LDI           -> CODELDI   :: ints
    | STI           -> CODESTI   :: ints
    | GETBP         -> CODEGETBP :: ints
    | GETSP         -> CODEGETSP :: ints
    | INCSP m       -> CODEINCSP :: m :: ints
    | GOTO lab      -> CODEGOTO  :: getlab lab :: ints
    | IFZERO lab    -> CODEIFZERO :: getlab lab :: ints
    | IFNZRO lab    -> CODEIFNZRO :: getlab lab :: ints
    | CALL(m,lab)   -> CODECALL  :: m :: getlab lab :: ints
    | TCALL(m,n,lab) -> CODETCALL :: m :: n :: getlab lab :: ints
    | RET m         -> CODERET   :: m :: ints
```

```
    | PRINTI       -> CODEPRINTI :: ints
    | PRINTC       -> CODEPRINTC :: ints
    | LDARGS       -> CODELDARGS :: ints
    | STOP         -> CODESTOP  :: ints
    | ARRLEN       -> CODEARRLEN :: ints

(* Convert instruction list to int list in two passes:
   Pass 1: build label environment
   Pass 2: output instructions using label environment
 *)

let code2ints (code : instr list) : int list =
    let (_, labenv) = List.fold makelabenv (0, []) code
    let getlab lab = lookup labenv lab
    List.foldBack (emitints getlab) code []
```

# Machine.java

```
/* File MicroC/Machine.java
   A unified-stack abstract machine for imperative programs.
   sestoft@itu.dk * 2001-03-21, 2009-09-24

   To execute a program file using this abstract machine, do:

      java Machine <programfile> <arg1> <arg2> ...

   or, to get a trace of the program execution:

      java Machinetrace <programfile> <arg1> <arg2> ...

*/

import java.io.*;
import java.util.*;

class Machine {
  public static void main(String[] args)
    throws FileNotFoundException, IOException {
    if (args.length == 0)
      System.out.println("Usage: java Machine <programfile> <arg1> ...\n");
    else
      execute(args, false);
  }

  // These numeric instruction codes must agree with Machine.fs:

  final static int
    CSTI = 0, ADD = 1, SUB = 2, MUL = 3, DIV = 4, MOD = 5,
    EQ = 6, LT = 7, NOT = 8,
    DUP = 9, SWAP = 10,
    LDI = 11, STI = 12,
    GETBP = 13, GETSP = 14, INCSP = 15,
    GOTO = 16, IFZERO = 17, IFNZRO = 18, CALL = 19, TCALL = 20, RET = 21,
    PRINTI = 22, PRINTC = 23,
    LDARGS = 24,
    STOP = 25,
    ARRLEN = 26;
```

```
final static int STACKSIZE = 1000;

// Read code from file and execute it

static void execute(String[] args, boolean trace)
  throws FileNotFoundException, IOException {
  int[] p = readfile(args[0]);            // Read the program from file
  int[] s = new int[STACKSIZE];           // The evaluation stack
  int[] iargs = new int[args.length-1];
  for (int i=1; i<args.length; i++)       // Push commandline arguments
    iargs[i-1] = Integer.parseInt(args[i]);
  long starttime = System.currentTimeMillis();
  execcode(p, s, iargs, trace);        // Execute program proper
  long runtime = System.currentTimeMillis() - starttime;
  System.err.println("\nRan " + runtime/1000.0 + " seconds");
}

// The machine: execute the code starting at p[pc]

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 CSTI:
      s[sp+1] = p[pc++]; sp++; break;
    case ADD:
      s[sp-1] = s[sp-1] + s[sp]; sp--; break;
    case SUB:
      s[sp-1] = s[sp-1] - s[sp]; sp--; break;
    case MUL:
      s[sp-1] = s[sp-1] * s[sp]; sp--; break;
    case DIV:
      s[sp-1] = s[sp-1] / s[sp]; sp--; break;
    case MOD:
      s[sp-1] = s[sp-1] % s[sp]; sp--; break;
    case EQ:
      s[sp-1] = (s[sp-1] == s[sp] ? 1 : 0); sp--; break;
    case LT:
      s[sp-1] = (s[sp-1] < s[sp] ? 1 : 0); sp--; break;
    case NOT:
      s[sp] = (s[sp] == 0 ? 1 : 0); break;
    case DUP:
      s[sp+1] = s[sp]; sp++; break;
    case SWAP:
      { int tmp = s[sp]; s[sp] = s[sp-1]; s[sp-1] = tmp; } break;
    case LDI:               // load indirect
      s[sp] = s[s[sp]]; break;
    case STI:               // store indirect, keep value on top
      s[s[sp-1]] = s[sp]; s[sp-1] = s[sp]; sp--; break;
    case GETBP:
      s[sp+1] = bp; sp++; break;
    case GETSP:
      s[sp+1] = sp; sp++; break;
```

```
    case INCSP:
      sp = sp+p[pc++]; break;
    case GOTO:
      pc = p[pc]; break;
    case IFZERO:
      pc = (s[sp--] == 0 ? p[pc] : pc+1); break;
    case IFNZRO:
      pc = (s[sp--] != 0 ? p[pc] : pc+1); break;
    case CALL: {
      int argc = p[pc++];
      for (int i=0; i<argc; i++)      // Make room for return address
        s[sp-i+2] = s[sp-i];          // and old base pointer
      s[sp-argc+1] = pc+1; sp++;
      s[sp-argc+1] = bp; sp++;
      bp = sp+1-argc;
      pc = p[pc];
    } break;
    case TCALL: {
      int argc = p[pc++];             // Number of new arguments
      int pop = p[pc++];              // Number of variables to discard
      for (int i=argc-1; i>=0; i--)   // Discard variables
        s[sp-i-pop] = s[sp-i];
      sp = sp - pop; pc = p[pc];
    } break;
    case RET: {
      int res = s[sp];
      sp = sp-p[pc]; bp = s[--sp]; pc = s[--sp];
      s[sp] = res;
    } break;
    case PRINTI:
      System.out.print(s[sp] + " "); break;
    case PRINTC:
      System.out.print((char)(s[sp])); break;
    case LDARGS:
      for (int i=0; i<iargs.length; i++) // Push commandline arguments
        s[++sp] = iargs[i];
      break;
    case STOP:
      return sp;
    case ARRLEN:
          s[sp] = s[sp] - s[s[sp]];
          break;
    default:
      throw new RuntimeException("Illegal instruction " + p[pc-1]
                                 + " at address " + (pc-1));
    }
  }
}

// Print the stack machine instruction at p[pc]

static String insname(int[] p, int pc) {
  switch (p[pc]) {
  case CSTI:  return "CSTI " + p[pc+1];
  case ADD:   return "ADD";
  case SUB:   return "SUB";
  case MUL:   return "MUL";
  case DIV:   return "DIV";
```

```
    case MOD:   return "MOD";
    case EQ:    return "EQ";
    case LT:    return "LT";
    case NOT:   return "NOT";
    case DUP:   return "DUP";
    case SWAP:  return "SWAP";
    case LDI:   return "LDI";
    case STI:   return "STI";
    case GETBP: return "GETBP";
    case GETSP: return "GETSP";
    case INCSP: return "INCSP " + p[pc+1];
    case GOTO:  return "GOTO " + p[pc+1];
    case IFZERO: return "IFZERO " + p[pc+1];
    case IFNZRO: return "IFNZRO " + p[pc+1];
    case CALL:  return "CALL " + p[pc+1] + " " + p[pc+2];
    case TCALL: return "TCALL " + p[pc+1] + " " + p[pc+2] + " " + p[pc+3];
    case RET:   return "RET " + p[pc+1];
    case PRINTI: return "PRINTI";
    case PRINTC: return "PRINTC";
    case LDARGS: return "LDARGS";
    case STOP:  return "STOP";
    case ARRLEN: return "ARRLEN";
    default:    return "<unknown>";
    }
  }

  // Print current stack and current instruction

  static void printsppc(int[] s, int bp, int sp, int[] p, int pc) {
    System.out.print("[ ");
    for (int i=0; i<=sp; i++)
      System.out.print(s[i] + " ");
    System.out.print("]");
    System.out.println("{" + pc + ": " + insname(p, pc) + "}");
  }

  // Read instructions from a file

  public static int[] readfile(String filename)
    throws FileNotFoundException, IOException
  {
    ArrayList<Integer> rawprogram = new ArrayList<Integer>();
    Reader inp = new FileReader(filename);
    StreamTokenizer tstream = new StreamTokenizer(inp);
    tstream.parseNumbers();
    tstream.nextToken();
    while (tstream.ttype == StreamTokenizer.TT_NUMBER) {
      rawprogram.add(new Integer((int)tstream.nval));
      tstream.nextToken();
    }
    inp.close();
    final int programsize = rawprogram.size();
    int[] program = new int[programsize];
    for (int i=0; i<programsize; i++)
      program[i] = ((Integer)(rawprogram.get(i))).intValue();
    return program;
  }
}
```

```
// Run the machine with tracing: print each instruction as it is executed

class Machinetrace {
  public static void main(String[] args)
    throws FileNotFoundException, IOException {
    if (args.length == 0)
      System.out.println("Usage: java Machinetrace <programfile> <arg1> ...\n");
    else
      Machine.execute(args, true);
  }
}
```