KØBENHAVNS UNIVERSITET, DATALOGISK INSTITUT

A Compiler for the FASTO Language

Allan Nielsen, Christian Nielsen, Troels Kamp Leskes

December 22, 2014

CONTENTS

1 Multiplication, division, boolean operators and literals 2

2 Filter and scan 3
- 2.1 Filter 3
  - 2.1.1 Testing 3
- 2.2 Scan 4
  - 2.2.1 Testing 6

3 λ-expressions in soacs 7

4 Copy propagation and constant folding 8

1 Multiplication, division, boolean operators and literals

Implementing multiplication and division was a simple matter, when having the already implemented code for addition and subtraction to look at. They served as a great way of getting to know the fasto compiler, and how things operate.
Negation is implemeted using the Mips.SUB instruction, where we pass the original argument to the operator, and subtracts this from zero.
So the instruction looks like:

```
Mips.SUB(place, "0", t1),
```

where t1 register that holds the argument x for ~x, and place is the register in which we place the result.
Not was more complicated than the previous ones, given that this requires more than one instruction to execute. However, the pattern learned here, proved to be useful for implementing and and or as well.

```
[ Mips.LI (place,"0")
, Mips.BNE (b,"0",falseLabel)
, Mips.LI (place,"1")
, Mips.LABEL falseLabel ]
```

Place is the register in which we want to store our result. We start by putting 0 into place, we then check if our argument b, actually is 0. Since Mips.BNE branches if its arguments are not equal, we will jump to falseLabel if and only if our argument b is 1, thus ending with a 0 in the place register, given we never execute Mips.LI(place, "1").
For boolean literals we've added for "true" and "false"
In our associativity we have the following code:

```
%nonassoc ifprec letprec
%left DEQ LTH
%left OR
%left AND
%nonassoc NOT
%left PLUS MINUS
%left TIMES DIV
%nonassoc NEG
```

Because of this hopefully we are getting the right bindings in our expressions. In the associations we made the %prec ifprec addition to the *if then else* and also the *not* statements. Because of this we are able to evaluate an expression like *not 4 == 2* as *not (4 == 2)* even though the *not* operator have a tighter associative binding than ==. We have been testing the associativity in two test cases *assoc.fo* and *assoc2.fo* to ensure that the bindings are right for all of our operators.

2 Filter and scan

2.1 Filter

For filter we expect a function with a return type of bool, and some type of array. We make sure this is the case in our typechecker, raise errors if we get a non-array argument or a non-bool function. If everything checks up, we among other things pass the type of the array to the code generator.

Given that filter takes some function and an array as input, and runs this function over the array, we notice that filter and map are similar in ways, so we based the code for filter on the code for map. With the difference being, that we do not want to store some computed value, instead we want to store the original value of the element we might be iterating on, only of the input function returns true.

We acheived this by making changes to the load part, so that we now store the actual value (instead of the computed one), in a register.
Since our input and output arrays will be of different sizes, we created a new counter *j_reg*, which gets incremented only when the input function on an element from the array, returns true.

Between the load and save sequence of map, we made a *check_fun* instruction set;

```
val check_fun = [ Mips.BEQ (res_reg, "0", loop_beg)
                , Mips.ADDI (j_reg, j_reg, "1") ]
```

where *res_reg* is the register in which we place the result of the function call on the input function. We branch to the loop beginning, if the result is false.

2.1.1 Testing

We used several tests for filter, making sure we use different types, as well as regular and lambda functions.
A test with a regular function, on integers.

```
fun bool great(int n) = 5 < n

fun int writeInt(int n) = write(n)

fun [int] main() =
    let a = filter(great, {6, 9, 8, 2}) in
    map(writeInt, a)
```

Similar test to the above, but with a lambda function, on integers.

```
fun int writeInt(int n) = write(n)

fun [int] main() =
    let a = filter(fn bool (int x) => 5 < x, {6, 9, 8, 2, ~5}) in
    map(writeInt, a)
```

A test with a lambda function on bools.

```
fun bool writeBool(bool s) = write(s)

fun [bool] main() =
  let a = {true, false, false, true} in
  let b = filter(fn bool (bool x) => not x, a) in
  map(writeBool, b)
```

All three tests perform as expected.

2.2 Scan

To implement `scan`, we wanted to reuse code from map and reduce, by combining the iterator from map and the accumulator from reduce in the code-generator.
In the typechecker, we pretty much reused the whole thing from reduce, adding that it should return an array of the return-type of the given function argument.

Below is the changed snippet for type-checking for `scan`:

```
...
              then (Array elem_type,
                        Out.Scan (f', n_dec, arr_dec, elem_type, pos))
...
```

The idea to implement this, was to accumulate the values using the given function into a placeholder and put that into the return-array as the function progress. Next thing was how to get the initial value into the first index of the return-array, which we found intuitively had to be done before the whole loop-process had begun.
The initializing code ended up like this:

```
...
val init_regs = [ Mips.ADDI (addr_reg, place, "4") (* point to the next word *)
                , Mips.SW (acc_reg, addr_reg, "0")
                , Mips.ADDI (addr_reg, addr_reg, "4") (* point to next word*)
                , Mips.MOVE(i_reg, "0") (* initialize iterator*)
                , Mips.ADDI (arr_reg, arr_reg, "4") (* Look at first element in input
    array *)]
```

```
...
```

For the looping itself, we split it into 5 parts, for the sole reason we found it easier to debug this way.
A header, that would check if the condition is satisfied:

```
...
            val loop_head =
                [ Mips.LABEL(loop_beg)
                , Mips.SUB(tmp_reg, i_reg, out_size_reg) (* make statement*)
                , Mips.BGEZ(tmp_reg, loop_end) ] (* if statement is equal to zero, jump
    to end *)
...
```

An upperbody, that loads the value of the next element in the input array to be evaluated:

```
...
val load_value =
  case getElemSize elem_type of
    One =>  [ Mips.LB   (elem_reg, arr_reg, "0")
            , Mips.ADDI (arr_reg, arr_reg, "1") ]
  | Four => [ Mips.LW   (elem_reg, arr_reg, "0")
            , Mips.ADDI (arr_reg, arr_reg, "4") ]
...
```

Next is where the magic happens, the application of the function on the accumulator/placeholder along with the newly loaded element. This is, as mentioned, put into the placeholder for next iteration:

```
...
val apply_code =
    applyFunArg(farg, [acc_reg, elem_reg], vtable, acc_reg, pos)
...
```

Afterwards we want to save this into the new array and count up the address register for next iteration.

```
...
val save_value =
  case getElemSize elem_type of
    One => [ Mips.SB (acc_reg, addr_reg, "0")
           , Mips.ADDI (addr_reg, addr_reg, "1")]
 | Four => [ Mips.SW (acc_reg, addr_reg, "0")
```

```
            , Mips.ADDI (addr_reg, addr_reg, "4")]
...
```

Last, but not least we increment the iterator and jump to the header for the next iteration:

```
...
val loop_foot = [ Mips.ADDI(i_reg, i_reg, "1")
                , Mips.J loop_beg
                , Mips.LABEL loop_end ]
```

2.2.1 Testing

When testing this, we found it does not work properly with bools. A simple test as this:

```
fun bool and(bool a, bool b) = a && b
fun bool or(bool a, bool b) = a || b

fun bool writeBool(bool n) = write(n)

fun [bool] main() =
  let a = {true, true, true, true, true, true, true, true} in
  let b = scan(or, true, a) in
  map(writeBool, b)
```

Yields the output `TrueFalseFalseFalseTrue...True` depending how big you make the array with true, but the 2nd to 4th always stays false. We believe it might have something to do in our code-generator some byte offset not being set porperly, but is somehow corrected when the iteration has run a few times.

For integers it works fine, as the test below yields what is expected.

```
fun int multi(int a, int b) = a * b

fun int writeInt(int n) = write(n)

fun [int] main() =
  let a = {1, 2, 3, 4} in
  let b = map(fn int (int x) => x*2, a) in
  let c = scan(multi, 1, b) in
  map(writeInt, c)
```

3 λ-EXPRESSIONS IN SOACS

To implement this we wanted to use as much from regular functions as possible, so we started going through each step, starting from the lexer, discussing how we could use already used code for function, to our advantage.

The lexing and parser part was quite simple, as we know λ-functions always begin with `fn` and the rest looks like a regular function, so we just pass everything on like a regular function. In the parser we put the whole thing under FunArgs, since λ-functions only are defined as being used in `map, filter, scan and reduce`.
Below is the `FunArg`-type shown

```
FunArg : ID { FunName (#1 $1) }
    | FN Type LPAR Params RPAR ARROW Exp
      { Lambda ( $2, $4, $7, $1 ) }
    | FN Type LPAR RPAR ARROW Exp
      { Lambda ( $2, [], $6, $1 ) }
;
```

As shown, these arguments are passed on to the `Lambda`-function in the typeChecker.
In the typeChecker, we create a `FunDec` using a dummy-name, as suggested. This is then passed to `checkFunWithVtable` where the function is typechecked, then the parameters are typechecked and its passed to the code-generator.
Below is the code for typechecking a `Lambda`-function shown:

```
| checkFunArg (In.Lambda (ret_type, params, body, funpos)
             , vtab, ftab, pos) =
    let val (Out.FunDec ( name, _, params, body', pos)) = checkFunWithVtable (In.
  FunDec ("Lambda", ret_type, params, body, pos), vtab, ftab, funpos)
      val arg_types = map (fn (Param (_, ty)) => ty) params
    in
    (Out.Lambda(ret_type, params, body', funpos), ret_type, arg_types)
    end
```

In the code-generator we need to bind the parameters with the arguments in the SymTab and do so recursively using a local defined function `bindVars`, that is only available in that scope. The base-cases is checking if one of the two arrays is empty while the other is not and throw and error if that is the case. Otherwise we want to return to updated vtable, once both the array of arguments and the array of parameters both are depleted simultaneously.
With the new vtable we then compute the expression for the `Lambda`-function.
Below is the code-generation for `Lambda` shown

```
 | applyFunArg (Lambda(ret_type, params, body', funpos), args, vtable, place, pos) :
  Mips.Prog =
    let
      fun bindVars ([], [], vtable) = vtable
        | bindVars([], args, vtable) = raise Error("stop det pjat", pos)
```

```
        | bindVars(params, [], vtable) = raise Error("stop det pjat stadigvæk", pos)
        | bindVars(Param (name, paramtype)::params, arg::args, vtable) = SymTab.bind
    name arg (bindVars(params, args, vtable))
      val newVtable = bindVars(params, args, vtable)
      val code1 = compileExp body' newVtable place
      in
        code1
      end
```

To test this, we simply created a test-case for each of the 4 SOACS that can use λ -functions.

Below is the test-case for λ-function in map SOAC. The tests for the other SOACs is found in the tests folder for our project.

```
fun int writeInt(int x) = write(x)

fun [int] writeIntArr([int] x) = map(writeInt, x)

fun int main() =
   let N = read(int) in
   let z = iota(N)    in
   let w = writeIntArr(map(fn int (int x) => x + 2, z)) in
   let nl = write("\n") in
   writeInt(reduce(op+, 0, w))
```

4 Copy propagation and constant folding

We have implemented *copy propagation and constant foldning* in our compiler, though it does not handle shadowing. Furthermore have all the features from task 1.
Below is the test-case for Times explained. This will indicate that both the cases work and it is able to actually optimize by propagating and fold correspondingly.

When we want to fold expressions, we can do so, by predicting what the result is going to be of a given expression and return this instead.
We thought through for each expression, what base-cases there could be and what we then should return.

Below is shown our code for the *Times*-expression.

```
| Times  (e1, e2, pos) =>
  let val e1' = copyConstPropFoldExp vtable e1
      val e2' = copyConstPropFoldExp vtable e2
  in case (e1', e2') of
         (Constant (IntVal x, _), Constant (IntVal y, _)) =>
         Constant (IntVal (x*y), pos)
       | (Constant (IntVal 0, _), _) =>
```

```
        e1'
      | (_, Constant (IntVal 0, _)) =>
        e2'
      | (Constant (IntVal 1, _), _) =>
        e2'
      | (_, Constant (IntVal 1, _)) =>
        e1'
      | _ =>
        Times (e1', e2', pos)
  end
```

We start by optimizing the two subexpressions. Then we check if any of the two expressions evaluates to 0 or 1. If one of them is 0, we return 0. Also, if one of them is 1 then it will be the other expression that is returned, no matter what it states. If none of the above applies, then we want to compute the expression.

Below is the code for testing *Times*.

```
fun int main() =
  let a = 5 in
  let b = a in
  let c = b in
   write(b + c * 0)
```

If our propagation is correct, all the variables should evaluate to 5, due to *a* having that constant assigned.
In the *write*-statement we have some expressions which also can be optimized. Since we multiply by 0, that whole expression will evaluate to 0. Then we plus *b* with 0 and end up with *b* as the result, and we end up writing *b*.
In the end, our program will look like this when optimized using copy propagation and constant folding.

```
fun int main() =
    let a = 5 in
    let b = 5 in
    let c = 5 in
    write(5)
```

APPENDIX

The following appendix shows the code we have added in the compiler phases Lexer.lex, Parser.grm, TypeChecker.sml, CodeGen.sml, Interpreter.sml and CopyConstPropFold.sml

Lexer.lex

```
  | "fn"            => Parser.FN pos
  | "not"           => Parser.NOT pos

  | "true"|"false"      { case Bool.fromString (getLexeme lexbuf) of
                              NONE   => lexerError lexbuf "Bad bool"
                            | SOME b => Parser.BOOLEAN (b, getPos lexbuf) }

  | `*`                 { Parser.TIMES (getPos lexbuf) }
  | `/`                 { Parser.DIV (getPos lexbuf) }
  | `~`                 { Parser.NEG (getPos lexbuf) }
  | "=>"                { Parser.ARROW (getPos lexbuf) }
  | "||"                { Parser.OR (getPos lexbuf) }
  | "&&"                { Parser.AND (getPos lexbuf) }
```

Parser.grm

```
%token <(int*int)> IF THEN ELSE LET IN INT BOOL CHAR EOF
%token <string*(int*int)> ID STRINGLIT
%token <int*(int*int)> NUM
%token <char*(int*int)> CHARLIT
%token <bool*(int*int)> BOOLEAN
%token <(int*int)> PLUS MINUS DEQ EQ LTH NEG NOT ARROW
%token <(int*int)> TIMES DIV AND NOT OR LPAR RPAR LBRACKET RBRACKET LCURLY RCURLY
%token <(int*int)> COMMA
%token <(int*int)> FUN FN IOTA REPLICATE MAP REDUCE FILTER SCAN READ WRITE
%token <(int*int)> OP

%nonassoc ifprec letprec
%left DEQ LTH
%left OR
%left AND
%nonassoc NOT
%left PLUS MINUS
%left TIMES DIV
%nonassoc NEG

        | Exp TIMES Exp
                        { Times($1, $3, $2) }
        | Exp DIV Exp
                        { Divide($1, $3, $2) }
        | NEG Exp
                        { Negate($2, $1) }
        | Exp AND Exp { And($1, $3, $2) }
        | Exp OR Exp
                        { Or($1, $3, $2) }

        | SCAN LPAR FunArg COMMA Exp COMMA Exp RPAR
                        { Scan ($3, $5, $7, (), $1)}
        | FILTER LPAR FunArg COMMA Exp RPAR
                        { Filter ($3, $5, (), $1) }

        | NOT Exp %prec ifprec { Not ( $2, $1 ) }

FunArg : ID { FunName (#1 $1) }
     | FN Type LPAR Params RPAR ARROW Exp
       { Lambda ( $2, $4, $7, $1 ) }
     | FN Type LPAR RPAR ARROW Exp
       { Lambda ( $2, [], $6, $1 ) }
;
```

TypeChecker.sml

```
    | In.Times (e1, e2, pos)
      => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, e1, e2)
          in (Int,
            Out.Times(e1_dec, e2_dec, pos))
          end

    | In.Divide (e1, e2, pos)
      => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, e1, e2)
          in (Int,
            Out.Divide(e1_dec, e2_dec, pos))
          end

    | In.Negate (e1, pos)
      => let val (t1, e1') = checkExp ftab vtab e1
         in (Int, Out.Negate(e1', pos))
         end

    | In.Not (e1, pos)
      => let val (t1, e1') = checkExp ftab vtab e1
         in (Bool, Out.Not(e1', pos))
         end

    | In.Or (e1, e2, pos)
      => let val (t1, e1') = checkExp ftab vtab e1
             val (t2, e2') = checkExp ftab vtab e2
             in
                if(t1 = Bool andalso t2 = Bool) then
                  (Bool, Out.Or(e1', e2', pos))
                else
                  raise Error("Type Error: Non-boolean arguments given to ||", pos)
          end

    | In.And (e1, e2, pos)
      => let val (t1, e1') = checkExp ftab vtab e1
             val (t2, e2') = checkExp ftab vtab e2
             in
                if(t1 = Bool andalso t2 = Bool) then
                  (Bool, Out.And(e1', e2', pos))
                else
                  raise Error("Type Error: Non-boolean arguments given to &&", pos)
          end

    | In.Scan (f, n_exp, arr_exp, _, pos) 256
      => let val (n_type, n_dec) = checkExp ftab vtab n_exp
             val (arr_type, arr_dec) = checkExp ftab vtab arr_exp
             val elem_type =
               case arr_type of
                  Array t => t
                | other => raise Error ("Scan: argument is not an array", pos)
             val (f', f_arg_type) =
               case checkFunArg (f, vtab, ftab, pos) of
                   (f', res, [a1, a2]) =>
```

```
                    if a1 = a2 andalso a2 = res
                    then (f', res)
                    else raise Error
                           ("Scan: incompatible function type of "
                            ^ In.ppFunArg 0 f ^": " ^ showFunType ([a1, a2], res), pos)
                  | (_, res, args) =>
                    raise Error ("Scan: incompatible function type of "
                                 ^ In.ppFunArg 0 f ^ ": " ^ showFunType (args, res), pos)
              fun err (s, t) =
                  Error ("Scan: unexpected " ^ s ^ " type " ^ ppType t ^
                         ", expected " ^ ppType f_arg_type, pos)
          in if elem_type = f_arg_type
             then if elem_type = n_type
                  then (Array elem_type,
                        Out.Scan (f', n_dec, arr_dec, elem_type, pos))
                  else raise (err ("neutral element", n_type))
             else raise err ("array element", elem_type)
          end

      | In.Filter (f, arr_exp, _, pos)  260
        => let val (arr_type, arr_exp_dec) = checkExp ftab vtab arr_exp

               val elem_type =
                 case arr_type of
                   Array t => t
                 | other => raise Error ("Filter: argument is not an array", pos)

               val (f', f_res_type, f_arg_type) =
                 case checkFunArg (f, vtab, ftab, pos) of
                     (f', Bool, [a1]) => (f', Bool, a1)
                   | (_,  res, args) =>
                    raise Error ("Filter: incompatible function type of "
                                 ^ In.ppFunArg 0 f ^ ":" ^ showFunType (args, res), pos)

               in (arr_type,
                        Out.Filter (f', arr_exp_dec, elem_type, pos))
              end

      | checkFunArg (In.Lambda (ret_type, params, body, funpos) 276
                    , vtab, ftab, pos) =
        let val (Out.FunDec ( name, _, params, body', pos)) = checkFunWithVtable (In.FunDec ("Lambda", ret_type, params, body, pos), vtab, ftab, funpos)
        val arg_types = map (fn (Param (_, ty)) => ty) params
        in
        (Out.Lambda(ret_type, params, body', funpos), ret_type, arg_types)
        end
```

```
      | Constant (BoolVal b, pos) =>
          if(b) then
            [Mips.LI(place, "1")]
          else
            [Mips.LI(place, "0")]

      | Times (e1, e2, pos) =>
          let val t1 = newName "times_L"
              val t2 = newName "times_R"
              val code1 = compileExp e1 vtable t1
              val code2 = compileExp e2 vtable t2
          in code1 @ code2 @ [Mips.MUL (place,t1,t2)]
          end
      | Divide (e1, e2, pos) =>
          let val t1 = newName "div_L"
              val t2 = newName "div_R"
              val code1 = compileExp e1 vtable t1
              val code2 = compileExp e2 vtable t2
          in code1 @ code2 @ [Mips.DIV (place,t1,t2)]
          end

      | Negate (e1, pos) =>
          let val t1 = newName "negateVal"
              val code1 = compileExp e1 vtable t1
          in code1 @ [ Mips.SUB(place, "0", t1) ]
          end

      | Not (b_exp, pos) =>
          let val b = "boolean"
              val code1 = compileExp b_exp vtable b
              val falseLabel = newName "false"
          in code1 @
              [ Mips.LI (place,"0")
              , Mips.BNE (b,"0",falseLabel)
              , Mips.LI (place,"1")
              , Mips.LABEL falseLabel ]
          end

      | Or (e1, e2, pos) =>
          let val trueLabel = newName "trueLabel"
              val falseLabel = newName "falseLabel"
              val endLabel = newName "endLabel"
              val code1 = compileCond e1 vtable trueLabel falseLabel
              val code2 = compileCond e2 vtable trueLabel endLabel
          in  [Mips.LI(place, "0")] @
                code1 @
                [Mips.LABEL falseLabel] @
                code2 @
                [Mips.LABEL trueLabel, Mips.LI(place, "1"), Mips.LABEL endLabel]
          end

      | And (e1, e2, pos) =>
```

```
        let val trueLabel = newName "trueLabel"
            val falseLabel = newName "falseLabel"
            val endLabel = newName "endLabel"
            val code1 = compileCond e1 vtable trueLabel falseLabel
            val code2 = compileCond e2 vtable endLabel falseLabel
        in  [Mips.ADD(place, "0", "1")] @
              code1 @
             [Mips.LABEL trueLabel] @
              code2 @
             [Mips.LABEL falseLabel, Mips.ADD(place, "0", "0"), Mips.LABEL endLabel]
        end

    (* Scan(f, e, [a1, a2, ..., an]) = [e, f(e, a1), f(f(e, a1), a2), ...] *)
  | Scan (farg, acc_exp, arr_exp, elem_type, pos) =>
        let val in_size_reg = newName "in_size_reg" (* size of input array *)
            val out_size_reg = newName "out_size_reg" (* size of ouput array *)
            val acc_reg = newName "acc_reg" (* last computed value for output *)
            val i_reg = newName "i_reg" (* Iterator register *)
            val addr_reg = newName "addr_reg" (* address of element in output array *)
            val arr_reg = newName "arr_reg"
            val elem_reg = newName "elem_reg"

            val arr_code = compileExp arr_exp vtable arr_reg
            val acc_code = compileExp acc_exp vtable acc_reg

            val get_size = [ Mips.LW (in_size_reg, arr_reg, "0") (* Loads array-size into in_size_reg *)
                           , Mips.ADDI(out_size_reg, in_size_reg, "1") (* Puts size into out_size_reg *)
                           ]

            (* Initiate registers.
               Put address of place into addr_reg, so we return proper addresses.
               Increment in_size_reg by 1 to determine out_size_reg, since output array is
               1 element longer. *)
            val init_regs = [ Mips.ADDI (addr_reg, place, "4") (* point to the next word *)
                            , Mips.SW (acc_reg, addr_reg, "0")
                            , Mips.ADDI (addr_reg, addr_reg, "4") (* point to next word*)
                            , Mips.MOVE(i_reg, "0") (* initialize iterator*)
                            , Mips.ADDI (arr_reg, arr_reg, "4") (* Look at first element in input array*)
                            ]


            val loop_beg = newName "loop_beg"
            val loop_end = newName "loop_end"
            val tmp_reg = newName "tmp_reg"

            (* while i_reg < in_size_reg*)
            val loop_head =
```

```
                [ Mips.LABEL(loop_beg)
                , Mips.SUB(tmp_reg, i_reg, out_size_reg) (* make statement*)
                , Mips.BGEZ(tmp_reg, loop_end) ] (* if statement is equal to zero, jump to end *)
            (* loads next value in input array into tmp_reg *)
            val load_value =
                case getElemSize elem_type of
                    One => [ Mips.LB   (elem_reg, arr_reg, "0")
                           , Mips.ADDI (arr_reg, arr_reg, "1") ]
                  | Four => [ Mips.LW   (elem_reg, arr_reg, "0")
                           , Mips.ADDI (arr_reg, arr_reg, "4") ]

            val apply_code =
                applyFunArg(farg, [acc_reg, elem_reg], vtable, acc_reg, pos)

            (* save the current accumulated value to a register for later computations,
              and to memory for later output *)
            val save_value =
                case getElemSize elem_type of
                    One => [ Mips.SB (acc_reg, addr_reg, "0")
                          , Mips.ADDI (addr_reg, addr_reg, "1")]
                  | Four => [ Mips.SW (acc_reg, addr_reg, "0")
                           , Mips.ADDI (addr_reg, addr_reg, "4")]



            (* increments i_reg *)
            val loop_foot =
                [ Mips.ADDI(i_reg, i_reg, "1")
                , Mips.J loop_beg
                , Mips.LABEL loop_end ]

        in [Mips.LABEL "Det_her_er_starten"]
           @ arr_code
           @[Mips.LABEL "array_kode"]
           @ acc_code
           @[Mips.LABEL "akku_kode"]
           @ get_size
           @[Mips.LABEL "Stoerrelsen_på_dyret"]
           @ dynalloc (out_size_reg, place, elem_type)
           @[Mips.LABEL "init_regz"]
           @ init_regs
           @[Mips.LABEL "starten_af_loopet"]
           @ loop_head
           @[Mips.LABEL "midten_af_loopet"]
           @ load_value
           @[Mips.LABEL "mere_midte"]
           @ apply_code
           @[Mips.LABEL "gem_den_akku"]
           @ save_value
           @[Mips.LABEL "foden_af_loopet"]
           @ loop_foot
           @ [Mips.LABEL "Det_her_er_slutningen_SCAN"]
```

```
        end

    (* filter(f(), acc, [a1,a2]) = [f(acc, a1), f(acc, a2)] *)
    | Filter (farg, arr_exp, elem_type, pos) =>
        let val size_reg = newName "size_reg" (* size of input/output array *)
            val arr_reg  = newName "arr_reg" (* address of input array *)
            val elem_reg = newName "elem_reg" (* address of single element *)
            val res_reg = newName "res_reg"
            val val_reg = newName "val_reg"
            val arr_code = compileExp arr_exp vtable arr_reg

            val get_size = [ Mips.LW (size_reg, arr_reg, "0") ] (* *)

            val addr_reg = newName "addr_reg" (* address of element in new array *)
            val i_reg = newName "i_reg"
            val j_reg = newName "j_reg"
            val init_regs = [ Mips.ADDI (addr_reg, place, "4") (*point to the next word, so we don't overwrite the size*)
                            , Mips.MOVE (i_reg, "0")
                            , Mips.LI (j_reg, "0")
                            , Mips.ADDI (elem_reg, arr_reg, "4") ]

            val loop_beg = newName "loop_beg"
            val loop_end = newName "loop_end"
            val tmp_reg = newName "tmp_reg"
            val loop_header = [ Mips.LABEL (loop_beg)
                              , Mips.SUB (tmp_reg, i_reg, size_reg)
                              , Mips.BGEZ (tmp_reg, loop_end)
                              , Mips.ADDI (i_reg, i_reg, "1") ]

            (* map is 'arr[i] = f(old_arr[i])'. *)
            val loop_load =
                case getElemSize elem_type of
                    One  => Mips.LB(val_reg, elem_reg, "0")
                              :: applyFunArg(farg, [val_reg], vtable, res_reg, pos)
                           @ [ Mips.ADDI(elem_reg, elem_reg, "1") ]
                  | Four => Mips.LW(val_reg, elem_reg, "0")
                              :: applyFunArg(farg, [val_reg], vtable, res_reg, pos)
                           @ [ Mips.ADDI(elem_reg, elem_reg, "4") ]

            val check_fun = [ Mips.BEQ (res_reg, "0", loop_beg)
                            , Mips.ADDI (j_reg, j_reg, "1") ]

            val loop_save =
                case getElemSize elem_type of
                    One => [ Mips.SB (val_reg, addr_reg, "0")]
                  | Four => [ Mips.SW (val_reg, addr_reg, "0") ]

            val loop_footer =
                [ Mips.ADDI (addr_reg, addr_reg,
                             makeConst (elemSizeToInt (getElemSize elem_type)))
                , Mips.J loop_beg
                , Mips.LABEL loop_end
                , Mips.SW (j_reg, place, "0")
```

```
                 ]
        in [Mips.LABEL "array_kode"]
          @ arr_code (* make arr_reg point to input array*)
          @[Mips.LABEL "measuring"]
          @ get_size (* gets the size of the input array *)
          @[Mips.LABEL "allocating_da_mem"]
          @ dynalloc (size_reg, place, elem_type) (* return place, which is an address where the ouput array is going to be *)
          @[Mips.LABEL "initializing"]
          @ init_regs
          @[Mips.LABEL "loopin"]
          @ loop_header
          @ loop_load
          @ check_fun
          @ [Mips.LABEL "saving"]
          @ loop_save
          @ loop_footer
        end

     | applyFunArg (Lambda(ret_type, params, body', funpos), args, vtable, place, pos) : Mips.Prog =
      let
        fun bindVars ([], [], vtable) = vtable
          | bindVars([], args, vtable) = raise Error("stop det pjat", pos)
          | bindVars(params, [], vtable) = raise Error("stop det pjat stadigvæk", pos)
          | bindVars(Param (name, paramtype)::params, arg::args, vtable) = SymTab.bind name arg (bindVars(params, args, vtable))
      val newVtable = bindVars(params, args, vtable)
      val code1 = compileExp body' newVtable place
      in
        code1
      end
```

Interpreter.sml

```
  | evalExp ( Times(e1, e2, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  evalBinopNum(op *, res1, res2, pos)
        end

  | evalExp ( Divide(e1, e2, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  evalBinopNum(op div, res1, res2, pos)
        end

  | evalExp ( Negate(e1, pos), vtab, ftab ) =
        let val res1 = evalExp(e1, vtab, ftab)
            val res2 = evalExp(Constant(IntVal 0, pos), vtab, ftab)
        in  evalBinopNum(op +, res1, res2, pos)
        end

  | evalExp ( Not(e1, pos), vtab, ftab ) =
        let val res1 = evalExp(e1, vtab, ftab)
        in  case res1 of
              BoolVal true => BoolVal false
            | BoolVal false => BoolVal true
            | _ => raise Error("Input is not of type bool", pos)
        end

  | evalExp ( And(e1, e2, pos), vtab, ftab ) =
        let val res1 = evalExp(e1, vtab, ftab)
            val res2 = evalExp(e2, vtab, ftab)
        in  case res1 of
              BoolVal false => res1
            | BoolVal true => (case res2 of
                                  BoolVal _ => res2
                                | _ => raise Error("Second argument is not of type bool", pos))
            | _ => raise Error("First argument is not of type bool", pos)

        end

  | evalExp ( Or(e1, e2, pos), vtab, ftab ) =
        let val res1 = evalExp(e1, vtab, ftab)
            val res2 = evalExp(e2, vtab, ftab)
        in  case res1 of
              BoolVal true => res1
            | BoolVal false => (case res2 of
                                  BoolVal _ => res2
                                | _ => raise Error("Second argument is not of type bool", pos))
            | _ => raise Error("First argument is not of type bool", pos)

        end
```

```
  | evalFunArg (Lambda(ret_type, params, exp, pos), vtab, ftab, callpos) =
    let
      val fundec = FunDec ("Lambda", ret_type, params, exp, pos)
    in
     (fn aargs => callFunWithVtable(fundec, aargs, vtab, ftab, callpos), ret_type)
    end
```

CopyConstPropFold.sml

```
fun copyConstPropFoldExp vtable e =
   case e of
       Constant x => Constant x
     | StringLit x => StringLit x
     | ArrayLit (es, t, pos) =>
       ArrayLit (map (copyConstPropFoldExp vtable) es, t, pos)
     | Var (name, pos) =>
       (* TODO TASK 4: This case currently does nothing.

        You must perform a lookup in the symbol table and if you find
        a Propagatee, return either a new Var or Constant node. *)
       let
         val value = SymTab.lookup name vtable
       in
         case value of
           SOME (VarProp v)   => copyConstPropFoldExp vtable (Var(v, pos))
         | SOME (ConstProp v) => copyConstPropFoldExp vtable (Constant(v, pos))
         | NONE               => Var(name, pos)
       end

   let val e' = copyConstPropFoldExp vtable e
      in
        let
          val vtable' =
            case e' of
              (Var (vname, p))      => (SymTab.bind name (VarProp(vname)) vtable)
            | (Constant (vval, p)) => (SymTab.bind name (ConstProp(vval)) vtable)
            | _ => vtable
        in
         Let (Dec (name, e', decpos), copyConstPropFoldExp vtable' body, pos)
        end
      end
```