Group Project Report in IPS

Oscar Zakaria Heiberg, `rmw874`
Valdemar Sander Ring, `fkn112`

June 8, 2023

Contents

Introduction

The structure of this report follows the idea of the different compiler phases. Through each of the project tasks, we will discuss the lexing/parsing, delve a little into the implementation of the interpreter, and lastly look at the code generation, and the type checking on which it relies on. After going through task 1 through 3, we will discuss the shortcomings of our code by evaluating our test suite.

1 Warm-Up

1.a Multiplication, division, boolean operators and literals

Lexer and Parser

The lexing and parsing needed to implement multiplication, division, boolean operators and literals was fairly straightforward, as they followed the same format. In the lexer we added keywords like the following:

```
let keyword (s, pos) =
     match s with
        ...
        | "true"         -> Parser.TRUE pos
        ...
```

And the lexer tokens were added just like the rest of the lexer tokens:

```
rule Token = parse
    ...
    | "&&"                  { Parser.AND (getPos lexbuf)}
    ...
```

In the parser, we then handle the output from the lexer by first recognizing the tokens like so:

```
...
%token <Position> MUL
...
```

We then have to make sure that functions of the highest precedence goes first, while also being sure of the associativity of the function. For most of the operators we implemented, we would prefer reading expressions from left to right, which resulted in almost all of our functions being left-associative. Below is the majority of the implementation:

```
%nonassoc ifprec letprec
%left OR
%left AND
%nonassoc NOT
```

```
...
%left PLUS MINUS
%right NEG
%left MUL DIV
...
```

Lastly, with the ground work laid out, we could send pass on our code to either the interpreter or typechecker by adding grammar rules for the expressions. As could be expected, most of the code for this part was also the same, but with minor tweaks. False is implemented like true, all other binops are implemented like multiplication, and unops are implemented like not:

```
Exp : ...
    | TRUE          { Constant (BoolVal true, $1)} //bool
    ...
    | Exp DIV   Exp  { Divide ($1, $3, $2)} //binop
    ...
    | NOT       Exp  { Not ($2, $1)} //unop
    ...
```

Interpreter

Moving on from the Lexer and Parser, we get to a point where we can interpret our code in F# instead of the Fasto language. Implementing the expressions in our interpreter was also a case of doing the same thing for each group of expression (groups being binary and unary operations). One special case would be division, as we would need to make sure we were not allowed to divide by zero.

```
let rec evalExp (e, vtab, ftab) =
  match e with
  ...
  | Divide(e1, e2, pos) -> //binop
      let res1  = evalExp(e1, vtab, ftab)
      let res2  = evalExp(e2, vtab, ftab)
      match (res1, res2) with
            | (IntVal n1, IntVal (n2: int)) ->
                if n2 = 0 then raise (MyError("Division by zero", pos))
                else IntVal (n1/n2)
            | (IntVal _, _) ->
                reportWrongType "right operand of /" Int res2 (expPos e2)
            | (_, _) ->
                reportWrongType "left operand of /" Int res1 (expPos e1)
    ...
  | Not(e1, pos) -> //unop
      let res1 = evalExp(e1, vtab, ftab)
      match res1 with
            | BoolVal b1 -> BoolVal (not b1)
            | _ -> reportWrongType "operand of not" Bool res1 (expPos e1)
    ...
```

Typechecker and CodeGen

From our short detour into setting up our code to be interpreted by F#, we move back to getting Fasto to work. Next step is to decorate our syntax tree with types. To do so, we use a type checker. As expected, all binops are essentially the same as other binops, and all unops are almost identical to other unops:

```
checkExp ftab vtab exp =
    match exp with
        ...
        | Divide (e1, e2, pos) -> //binop
        let (e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, e1, e2)
        (Int, Divide (e1_dec, e2_dec, pos))
        ...
        | Not (e1, pos) -> //unop
            let (t1, e1') = checkExp ftab vtab e1
            if t1 <> Bool then
              reportTypeWrong "argument of not" Bool t1 pos
            (Bool, Not (e1', pos))
        ...
```

For the binary operations, we have a helper function, checkBinOp, which ensures that the types of the two operands are both are of type `t`, which of course, also means they are of the same type. The unary operations are essentially handled the same way that binary operations are handled. The difference here being that the we only one type to check. With the types all in order, we move onto the machine code generation. We start off by letting true/false be represented by 1/0 by loading the corresponding immediate of the binary into place.

```
  ...
  | Constant (BoolVal p, _) -> //num representation
      [ LI (place, if p then 1 else 0) ]
  ...
  | Divide (e1, e2, pos) ->  //binop
      let t1 = newReg "divide_L"
      let t2 = newReg "divide_R"
      let divByZero = newLab "divByZero"
      let divLegal = newLab "divLegal"
      let code1 = compileExp e1 vtable t1
      let code2 = compileExp e2 vtable t2
      let div = [ BEQ (t2, Rzero, divByZero)
                ; DIV (place,t1,t2)
                ; J divLegal
                ; LABEL divByZero
                ; LA (Ra1, "m.DivZero")
                ; J "p.RuntimeError"
                ; LABEL divLegal
                ]
      code1 @ code2 @ div
  ...
  | Not (e1, pos) -> //unop
      let t1 = newReg "not"
```

```
    let code1 = compileExp e1 vtable t1
    code1 @ [XORI (place,t1,1)]
```

The code for the other binary operations follow the same structure as Divide, but instead of a check to ensure values are legal, the we simply execute the RISC-V instruction (since all values of the correct type are legal, and all type checking is done beforehand). We have left out the other functions, as there are simple RISC-V instructions to compute them.

1.b Multiple-declaration `lets`

Lexer and Parser

For the lexer, the only thing we did was implement semicolon as a lexer token just like we did for all other legal symbols:

```
rule Token = parse
  ...
  | ';'                  {Parser.SEMICOLON (getPos lexbuf)}
  ...
```

However, when we look at the parser, things start to get interesting. For a more general production of let, we have to use recursion. Instead of reading the let as is, we now read the first part of the let, and then check if the body contains a semicolon, or if it contains the keyword "IN". In the case it contains a Semicolon, we recursively go through the helper function called "Lets" until we get the IN keyword. The implementation looks like the following:

```
Exp :...
    | LET ID EQ Exp Lets %prec letprec { Let (Dec (fst $2, $4, $3), $5, $1)}
;
Lets : SEMICOLON ID EQ Exp Lets { Let (Dec (fst $2, $4, $3), $5, $1) }
     | IN Exp                          { $2 }
;
```

Interpreter, Typechecker and CodeGen

Let was already implemented in the interpreter, typechecker and in the codegen, and did not need to be updated to accomodate for the new implementation in the parser.

2 Implementing replicate, filter and scan

2.a Replicate

Replicate takes a number n, and an input a, and replicates the input a, n times.

Lexer and Parser

Replicate, like filter, scan and the other functions is regarded as a keyword, which means we implement the lexer token as follows:

```
let keyword (s, pos) =
     match s with
       ...
       | "replicate"    -> Parser.REPLICATE pos
       ...
```

As mentioned, the same thing is done for filter and scan (with minor tweaks to accommodate for different names), hence why a code snippet will not be shown in the next sections. As for the parser, we decided the same set of functions should all be left-associative, and hold a fairly high precedence. We implemented the tokens and associativity like so:

```
...
%token <Position> DEQ LTH EQ MAP REDUCE IOTA ARROW REPLICATE SCAN FILTER
...
%left MAP REDUCE FILTER SCAN REPLICATE IOTA
...
```

Since replicate takes two expressions as an input, and those expressions are separated by a comma, we want our parser to recognize that. Whenever the keyword "Replicate" appears in front of a parenthesis-pair enclosing two expressions separated by a comma, we want to call the replicate function.

```
Exp :...
    | REPLICATE LPAR Exp COMMA Exp RPAR { Replicate ($3, $5, (), $1) }
```

Interpreter

With lexing and parsing handled, we can now move on to the interpreter. We wanted to first make sure that we respected the rule of n being at least 0. We start off by loading in the size and a, as well as the type of a. This lets us check if the size is of the correct type, and if so, if size is at least 0. In the case that either check is false, we simply raise an error, but if it is correct, we create a list of s elements, and convert them into an array of a's.

```
  | Replicate (n, a, atype, pos) ->
        let size = evalExp(n, vtab, ftab)
        let aval = evalExp(a, vtab, ftab)
        let atype = valueType aval
        match (size) with
          | IntVal s ->
              if s >= 0 then ArrayVal( List.map (fun x -> aval) [0..s-1], atype )
              else let msg = sprintf "Argument of \"replicate\" is negative: %i" s
                   raise (MyError(msg, pos))
          | _ -> reportWrongType "argument of \"replicate\"" Int size pos
```

Typechecker and CodeGen

Back to fasto. After creating our syntax tree, we want to decorate it with types. This is done in the type checker, and for replicate in specific, this is done by ensuring n is an int. If it is, we pass it on to the CodeGen step, and if not, we report an error.

```
| Replicate (e_exp, el_exp, _, pos) ->
    let (e_type, e_exp_dec) = checkExp ftab vtab e_exp
    let (el_type, el_exp_dec) = checkExp ftab vtab el_exp
    if e_type <> Int then reportTypeWrong "argument of replicate" Int e_type pos
    (Array el_type, Replicate (e_exp_dec, el_exp_dec, el_type, pos))
```

Now that we know n is an int, we can replicate whatever a is, n times. Actually, not yet, we still need to generate the code that needs to be run. The implementation of replicate had many similarities to the implementation of iota. Using this, we started out by defining some registers we knew we wanted to use, and using our vtable to compile expressions into registers. After doing so, we checked if n was greater than or equal to 0. The interesting part of the implementation is found the `loop_code`, codeblock. Here we initialize the registers and output array, and then we simply looped until `i_reg` was equal to n, storing a in `arr[i]`. For storing, we made sure to use the implemented Store function, s.t. we would avoid the problem of storing the . Later, when implementing filter and scan, we realized that it would likely be more maintainable if we initialized the code in another codeblock, and had a codeblock for both the header and footer of the loop. Lastly we run each of the code blocks in such an order, that we allocate space memory we use it.

```
| Replicate (n_exp, elem_exp, ret_type, pos) ->
    let arr_reg = newReg "arr"                        (* address of array *)
    let size_reg = newReg "size"
    let elem_reg = newReg "elem"
    let elem_code = compileExp elem_exp vtable elem_reg     (*compile elem_exp, which is the element to be
↪  replicated, into elem_reg*)
    let n_code = compileExp n_exp vtable size_reg     (* compile n_exp into size_reg *)
    let safe_lab = newLab "safe"                      (* safe label, used for checking if n is >= 0 *)
    let type_size = elemSizeToInt(getElemSize ret_type)
    let checksize = [ BGE (size_reg, Rzero, safe_lab) (* check if n >= 0 *)
                    ; LA (Ra1, "m.BadSize")           (* if not, raise error *)
                    ; J "p.RuntimeError"
                    ; LABEL (safe_lab)
                    ]
    let dynalloc_code = dynalloc (size_reg, arr_reg, ret_type)
    let set_first_elem = [ SW (size_reg, arr_reg, 0) ]
    let loop_beg = newLab "loop_beg"
    let loop_end = newLab "loop_end"
    let i_reg = newReg "i"
    let start_addr = [ MV(place, arr_reg) ]
    let loop_code = [ ADDI (arr_reg, arr_reg, 4) (* set arr_reg to address of first element instead *)
                    ; MV (i_reg, Rzero)          (* set i_reg to 0 *)
                    ; LABEL (loop_beg)
```

```
                    ; BGE (i_reg, size_reg, loop_end) (* while i < size_reg, loop *)
                    ; Store(getElemSize ret_type) (elem_reg, arr_reg, 0) (* arr[i] = a *)
                    ; ADDI (arr_reg, arr_reg, type_size) (* increment arr_reg by elem_size *)
                    ; ADDI (i_reg, i_reg, 1) (* increment i_reg by 1 *)
                    ; J loop_beg
                    ; LABEL (loop_end)
                    ]
    elem_code @
    n_code @
    checksize @
    dynalloc_code @
    start_addr @
    set_first_elem @
    loop_code
```

2.b Filter

Filter takes a predicate p and an array of type alpha, and returns a new array containing the elements for which the predicate holds.

Lexer and Parser

For the lexer and parser steps of filter, we essentially did the exact same thing as we do for replicate (see replicate for lexer). Filter takes only one Exp, but it also takes a FunArg; this, however, is not a problem, as we just pass it on for the typechecker and codegen to handle.

```
Exp :...
    | FILTER LPAR FunArg COMMA Exp RPAR { Filter ($3, $5, (), $1) }
;
FunArg : ID    { FunName (fst $1 ) } //not implemented by us, but relevant for Filter (and scan)
       | FN Type LPAR RPAR ARROW Exp { Lambda ($2, [], $6, $1) }
       | FN Type LPAR Params RPAR ARROW Exp { Lambda ($2, $4, $7, $1) }
;
```

Interpreter

In retrospect, it would be much easier to implement filter using the F# function, filter, however, when implementing the function, we did not think about that. Instead, we made a recursive function that takes each element from the array and check if the predicate is satisfied. If it returns true, we save the element, otherwise we move on.

```
    | Filter (farg, arrexp, t, pos) ->
        let arr = evalExp(arrexp, vtab, ftab)
        match arr with
        | ArrayVal (lst, tp1) ->
            let rec filter_helper lst =
                match lst with
                | [] -> []
                | x :: xs ->
```

```
                let result = evalFunArg (farg, vtab, ftab, pos, [x])
                if result = BoolVal true
                then x :: filter_helper xs
                else filter_helper xs
            let filter_result = filter_helper lst
            ArrayVal (filter_result, tp1)
        | otherwise -> reportNonArray "2nd argument of \"filter\"" arr pos
```

Typechecker and CodeGen

For checking the types of filter, we only really have to check that the input and output are of the same type, and that the predicate would result in a bool. The following shows our type checking:

```
    | Filter (f, arr_exp, _, pos) ->
        let (arr_type, arr_exp_dec) = checkExp ftab vtab arr_exp
        let elem_type =
            match arr_type with
              | Array t -> t
              | _ -> reportTypeWrongKind "second argument of filter" "array" arr_type pos
        let (f', f_res_type, f_arg_type) =
            match checkFunArg ftab vtab pos f with
              | (f', res, [a1]) -> (f', res, a1)
              | (_, res, args) ->
                    reportArityWrong "first argument of filter" 1 (args,res) pos
        if elem_type <> f_arg_type then
          reportTypesDifferent "function-argument and array-element types in filter"
                               f_arg_type elem_type pos
        if f_res_type <> Bool then
            reportTypeWrong "function-argument of filter" Bool f_res_type pos
        (Array elem_type, Filter (f', arr_exp_dec, elem_type, pos))
```

With types sorted, we can generate the code that our machine then reads and translates to binaries. Filter was when we realized that dividing our code into smaller chunks would be easier to read and debug. As usual, we created a bunch of obvious registers, but a not-so-obvious register would be `reg_res2` and a not-so-obvious label would be `falseLabel`. The second result register was used to hold the boolean result of the predicate, and the falseLabel was there to let us skip a bunch of code in the case that the predicate was false.

```
    | Filter (farg, arr_exp, ret_type, pos) ->
      let size_reg = newReg "size" (* size of input array. will be used to dynalloc, as output size is not
↪  known before running *)
      let arr_reg  = newReg "arr"  (* address of array *)
      let elem_reg = newReg "elem" (* address of current element *)
      let res_reg = newReg "res"   (* value to be stored*)
      let res_reg2  = newReg "res2" (* boolean result of applyfunarg*)
      let addr_reg = newReg "addrg" (* address of element in new array *)
      let i_reg = newReg "i" (* i for looping through input array *)
      let counter_reg = newReg "counter" (* counter for output array *)
      let loop_beg = newLab "loop_beg"
      let loop_end = newLab "loop_end"
```

```
      let falseLabel = newLab "false"
      let arr_code = compileExp arr_exp vtable arr_reg

      let init = [ ADDI (addr_reg, place, 4)   (* set addr_reg to address of first element instead *)
                 ; MV (i_reg, Rzero)           (* set i_reg to 0 *)
                 ; MV (counter_reg, Rzero)     (* set counter_reg to 0 *)
                 ; ADDI (elem_reg, arr_reg, 4) (* set elem_reg to address of first element instead *)
                 ]

      let get_size = [ LW (size_reg, arr_reg, 0) ] (* get size of input array. this will be what i loops over
↪  *)
      let src_size = getElemSize ret_type (* size of input array elements *)
      let dst_size = getElemSize ret_type (* size of output array elements *)
      let loop_header = [
                        LABEL (loop_beg)
                        ; BGE (i_reg, size_reg, loop_end) (* while i < size_reg, loop *)
                        ]

      let loop_filter = [
                        Load src_size (res_reg, elem_reg, 0)
                        ; ADDI (elem_reg, elem_reg, elemSizeToInt src_size) (* increment elem_reg by elem_size
↪  *)
                        ]
                        @ applyFunArg(farg, [res_reg], vtable, res_reg2, pos)  (* apply f to elem_reg, store
↪  result in addr_reg *)
                        @
                        [
                        BEQ (res_reg2, Rzero, falseLabel) (* if f(elem_reg) == 0, jump to falseLabel *)
                        ; Store dst_size (res_reg, addr_reg, 0)  (* store elem_reg in addr_reg *)
                        ; ADDI (addr_reg, addr_reg, elemSizeToInt dst_size) (* increment res_reg by elem_size
↪  *)
                        ; ADDI (counter_reg, counter_reg, 1) (* increment counter_reg by 1 *)
                        ; LABEL (falseLabel)
                        ]

      let loop_footer = [
                        ADDI (i_reg, i_reg, 1) (* increment i_reg by 1 *)
                        ; J loop_beg
                        ; LABEL (loop_end)
                        ]

      let set_size = [ SW (counter_reg, place, 0) ] (* set size of output array to counter_reg *)
      arr_code
      @ get_size
      @ dynalloc (size_reg, place, ret_type)
      @ init
      @ loop_header
      @ loop_filter
      @ loop_footer
      @ set_size
```

2.c Scan

Scan is the first of our functions that take three input. A binary operation, an accumulator and an array (on which we apply the binop with the accumulator)

Lexer and Parser

See replicate for lexer. For the parser, we now need to take care of three inputs. That is, we will have a binop as a FunArg, an Exp for the accumulator and an Exp for the array. We essentially did the same as we did for filter, but added another Exp before closing the parenthesis.

```
Exp :...
    | SCAN LPAR FunArg COMMA Exp COMMA Exp RPAR
                    { Scan ($3, $5, $7, (), $1) }
```

Interpreter

For the interpreter implementation of `Scan`, we firstly designed the function around having an array as input and also returning an array, therefore the concatenation. The we needed to make sure that the accumulator was updated each index, such that if fx the farg is plus, we can use i-1 as the other value than i, that should be added.

```
    | Scan (farg, ne, arrexp, tp, pos) ->
        let arr = evalExp(arrexp, vtab, ftab)
        let nel = evalExp (ne, vtab, ftab)
        match arr with
        | ArrayVal (lst, tp1) ->
            let initial_acc = nel
            let rec scan_helper acc lst =
                match lst with
                | [] -> []
                | x :: xs ->
                    let result = evalFunArg (farg, vtab, ftab, pos, [acc; x])
                    let new_acc = result
                    let rest = scan_helper new_acc xs
                    new_acc :: rest
            let scan_result = scan_helper initial_acc lst
            ArrayVal (scan_result, tp1)
        | _ -> reportNonArray "3rd argument of \"scan\"" arr pos
```

Typechecker and CodeGen

For the implementation of the typechecker for `Scan`, we essentially did what we did in the two other typechecker cases, by ensuring that each element was of the correct type, the accumulator was of the correct type such that we would be apply to apply our binary operation to both.

```
    | Scan (f, e_exp, arr_exp, _, pos) ->
        let (e_type  , e_dec  ) = checkExp ftab vtab e_exp
        let (arr_type, arr_dec) = checkExp ftab vtab arr_exp
        let elem_type =
            match arr_type with
              | Array t -> t
              | _ -> reportTypeWrongKind "third argument of scan" "array" arr_type pos
        let (f', f_argres_type) =
            match checkFunArg ftab vtab pos f with
              | (f', res, [a1; a2]) ->
                  if a1 <> a2 then
                     reportTypesDifferent "argument types of operation in scan"
                                          a1 a2 pos
                  if res <> a1 then
                     reportTypesDifferent "argument and return type of operation in scan"
                                          a1 res pos
                  (f', res)
              | (_, res, args) ->
                  reportArityWrong "operation in scan" 2 (args,res) pos
        if elem_type <> f_argres_type then
          reportTypesDifferent "operation and array-element types in scan"
                               f_argres_type elem_type pos
        if e_type <> f_argres_type then
          reportTypesDifferent "operation and start-element types in scan"
                               f_argres_type e_type pos
        (Array elem_type, Scan (f', e_dec, arr_dec, elem_type, pos))
```

Now onto the code generation for `Scan`. Every line has been commented, to make it more comprehensible to understand where different functionality lies. We've create different loop parts (header, filter and footer), where filter contains the main loop and the header contains the condition for the loop and the footer updates variables, to get the next iteration ready.

```
  | Scan (binop, acc_exp, arr_exp, tp, pos) ->
      let size_reg = newReg "size" (* size of input array. will be used to dynalloc, as output size is not
↪  known before running *)
      let arr_reg  = newReg "arr"  (* address of array *)
      let elem_reg = newReg "elem" (* address of current element *)
      let res_reg = newReg "res"   (* value loaded *)
      let res_reg2  = newReg "res2" (* value stored *)
      let addr_reg = newReg "addrg" (* address of element in new array *)
      let tmp_reg = newReg "tmp" (* address of element in new array *)
      let i_reg = newReg "i" (* i for looping through input array *)
      let loop_beg = newLab "loop_beg"
      let loop_end = newLab "loop_end"
      let acc_code = compileExp acc_exp vtable res_reg
      let arr_code = compileExp arr_exp vtable arr_reg
      let init = [ ADDI (addr_reg, place, 4)   (* set addr_reg to address of first element instead *)
                 ; MV (i_reg, Rzero)           (* set i_reg to 0 *)
                 ; ADDI (elem_reg, arr_reg, 4) (* set elem_reg to address of first element instead *)
                 ]
      let get_size = [ LW (size_reg, arr_reg, 0) ] (* get size of input array. this will be what i loops over
↪  *)
      let src_size = getElemSize tp (* size of input array elements *)
      let dst_size = getElemSize tp (* size of output array elements *)
```

```
      let loop_header = [
                        LABEL (loop_beg)
                        ; BGE (i_reg, size_reg, loop_end) (* while i < size_reg, loop *)
                        ]
      let loop_filter = [
                        Load src_size (res_reg2, elem_reg, 0)  (* load elem_reg from addr_reg *)
                        ; ADDI (elem_reg, elem_reg, elemSizeToInt src_size) (* increment elem_reg by src_size
↪  *)
                        ]
                        @ applyFunArg(binop, [res_reg; res_reg2], vtable, res_reg, pos)
                        @
                        [
                        Store dst_size (res_reg, addr_reg, 0) (* store res_reg2 in addr_reg *)
                        ; ADDI (addr_reg, addr_reg, elemSizeToInt dst_size) (* increment addr_reg by dst_size
↪  *)
                        ]

      let loop_footer = [
                        ADDI (i_reg, i_reg, 1) (* increment i_reg by 1 *)
                        ; Load dst_size (tmp_reg, addr_reg, -elemSizeToInt src_size) (* store previous value in
↪  tmp. done at this stage to avoid the size element *)
                        ; J loop_beg
                        ; LABEL (loop_end)
                        ]
      let set_size = [ SW (size_reg, place, 0) ] (* set size of output array to size_reg *)

      arr_code
      @ get_size
      @ dynalloc (size_reg, place, tp)
      @ init
      @ acc_code
      @ loop_header
      @ loop_filter
      @ loop_footer
      @ set_size
```

3 High-level optimizations

3.a Copy propagation, constant folding, and dead-binding removal

Copy Propagation, Constant Folding, Variable shadowing

Now that we have code that runs, we want to make it run even faster. To do so, we implement copy propagation and constant folding. With copy propagation we, for example, check if a variable is in the table. In the case it is, we return the value instead of the variable. This is done by the implementation of the recursive function, copyConstPropFoldExp. Implementing the case for a var is fairly simple:

```
| Var (name, pos) ->
    let e' = SymTab.lookup name vtable
```

```
    match e' with
        | Some (VarProp v) -> Var (v, pos)
        | Some (ConstProp c) -> Constant (c, pos)
        | _ -> Var(name, pos)
```

For optimizing index, we can check if the name of the array already exists in the table, and if it does, we can optimize the index by calling the copyConst function recursively:

```
| Index (name, e, t, pos) ->
            let e' = SymTab.lookup name vtable
            let e'' = copyConstPropFoldExp vtable e // we do this to optimize the index expression
            match e' with
                | Some (VarProp v) -> Index (v, e'', t, pos)
                | _ -> Index (name, e'', t, pos)
```

When optimizing lets, things start to get interesting. The body of the let can either be a variable, a constant or another let. We have to account for all of these cases. The variable and constant is basically the same thing with VarProp and ConstProp being the only difference:

```
| Var (name, pos) -> // x = y
    let vtable' = SymTab.bind name (VarProp name) vtable
    let body' = copyConstPropFoldExp vtable' body
    body'
```

But when we look into the last case with let, we have some more things to consider. As presented in the hint we want to rearrange the input in such a way that the inner let is expressed first. We did this by calling copyConst on two new let declarations with their order switched around:

```
| Let (Dec (name2, e2, decpos2), body2, pos2) ->
    let body' = copyConstPropFoldExp vtable (
        Let(Dec(name2, e2, decpos),
        Let(Dec(name, body2, decpos2), body, pos2), pos))
    body'
```

With the hard part done, we can look at the simpler parts, i.e. the case of Times and And. Here we started out running copyConst on our two expressions, which, trivially, would give us two numbers. From this, we could would implement some safe algebraic simplifications:

```
| Times (e1, e2, pos) ->
    let e1' = copyConstPropFoldExp vtable e1
    let e2' = copyConstPropFoldExp vtable e2
    match (e1', e2') with
        | (Constant (IntVal x, _), Constant (IntVal y, _)) -> Constant (IntVal (x * y), pos)
        | (Constant (IntVal 1, _), _) -> e2'
        | (_, Constant (IntVal 1, _)) -> e1'
        | (Constant (IntVal 0, _), _) ->  Constant (IntVal 0, pos)
        | (_, Constant (IntVal 0, _)) -> Constant (IntVal 0, pos)
        | (_, Constant (IntVal -1, _)) -> Minus(Constant(IntVal 0, pos), e1', pos)
        | (Constant (IntVal -1, _), _) -> Minus(Constant(IntVal 0, pos), e2', pos)
        | _ -> Times (e1', e2', pos)
```

The same thing was done for And (although we omitted the case of true and true being true, as we got a compilation warning for doing so).

```
| And (e1, e2, pos) ->
    let e1' = copyConstPropFoldExp vtable e1
    let e2' = copyConstPropFoldExp vtable e2
    match (e1', e2') with
        | (Constant (BoolVal a, _), Constant (BoolVal b, _)) -> Constant (BoolVal (a && b), pos)
        | (Constant (BoolVal false, _), _) -> Constant (BoolVal false, pos)
        | (_, Constant (BoolVal false, _)) -> Constant (BoolVal false, pos)
        | _ -> And (e1', e2', pos)
```

We did not do the optional task of handling shadowing. This, in turn, does cause shadowing problems when we run the optimizations on our tests. A possible fix would entail changing the symbol table we use to keep track of variable names and their constants or variables.

Dead-binding removal

Now for removing bindings that are dead. We start out with the Variable. No matter what, a variable will never contain IO. Given the hints, we can implement variable by setting IO to false, we register the variable using recordUse on an empty symtab, and then we just return the expression:

```
| Var (name, pos) ->
    (false, (recordUse name (SymTab.empty())), Var (name, pos))
```

The same goes for index, or at least almost. We start out by calling removeDeadBindingsIn-Exp on the index of the array, and by doing so, we have information about IO, uses and we have the optimized version of the index. In doing so, we can simply return the boolean value of the io, record the name into uses, and then call the optimized index:

```
| Index (name, e, t, pos) ->
    let (eio, euses, e') = removeDeadBindingsInExp (e)
    (eio, (recordUse name euses), Index (name, e', t, pos))
```

Lastly, as to be expected, Let is a little more tedious. We know that if either the expression has IO or if the name is used in buses, we need to combine the uses from expression with the uses from body in order to reach optimal code. In the case that neither is true, we still need to remove the binding from the symtab. In code, it looks like this:

```
| Let (Dec (name, e, decpos), body, pos) ->
    let (eio, euses, e') = removeDeadBindingsInExp e
    let (bio, buses, body') = removeDeadBindingsInExp body
    if ((isUsed name buses) || eio) then
        (eio || bio,
        SymTab.combine euses (SymTab.remove name buses),
        Let (Dec (name, e', decpos), body', pos))
    else (bio, buses, body')
```

4 Evaluation

4.a Testing

In addition to the test suite handed out in the fasto.zip, we've created more nontrivial tests for the new functionality that has been implemented. This has been done, as we want to evaluate the program and find the cases in which our program fail (if any). The testing method is unit testing and more specifically blackbox-testing. Blackbox-testing has been used, as we just need a general overview on the correctness of our implementation. It is not essential for our compiler, that each branch in each method is handling data correctly, as long as input results in expected output (which almost implies correct data handling in the branches). To our already existing test suite, we added tests to ensure that functions would work with a variety of different (allowed) types, and that our basic operators such as not, would correctly "not" the boolean input.

4.b Shortcomings

In general, there haven't been a lot of shortcomings, at least any that have been found, in our test suite or implementation. All tests, except for the tests run with the optimization passes and the test aimed at array comprehension. An error we did catch, but was only partly an error happens when we divide by zero. We return an error as expected, however, the position of the error is not returned properly. When turning the optimizations on, we run in to a bunch of errors. Most of the error messages state that some Name was not found at line x, column y. This tells us that the majority of our errors are due to variable shadowing, however, some of the errors could potentially be attributed to some other misunderstanding in our removal of dead bindings.

5 Conclusion

From our test suite, we know that our implementation of the compiler works for the fasto language. We had some shortcomings that could be fixed by finding solutions to the bonus tasks, however, lack of time made this a challenge. Throughout the implementation of the compiler, we generally took inspiration from other functions, tokens etc. already implemented. This way we got a good understanding of the structure of a compiler and also how, more specifically, a compiler for fasto can be created with F#. We are quite confident in our implementation, as the implementation succeeds a pretty thorough test suite.