FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

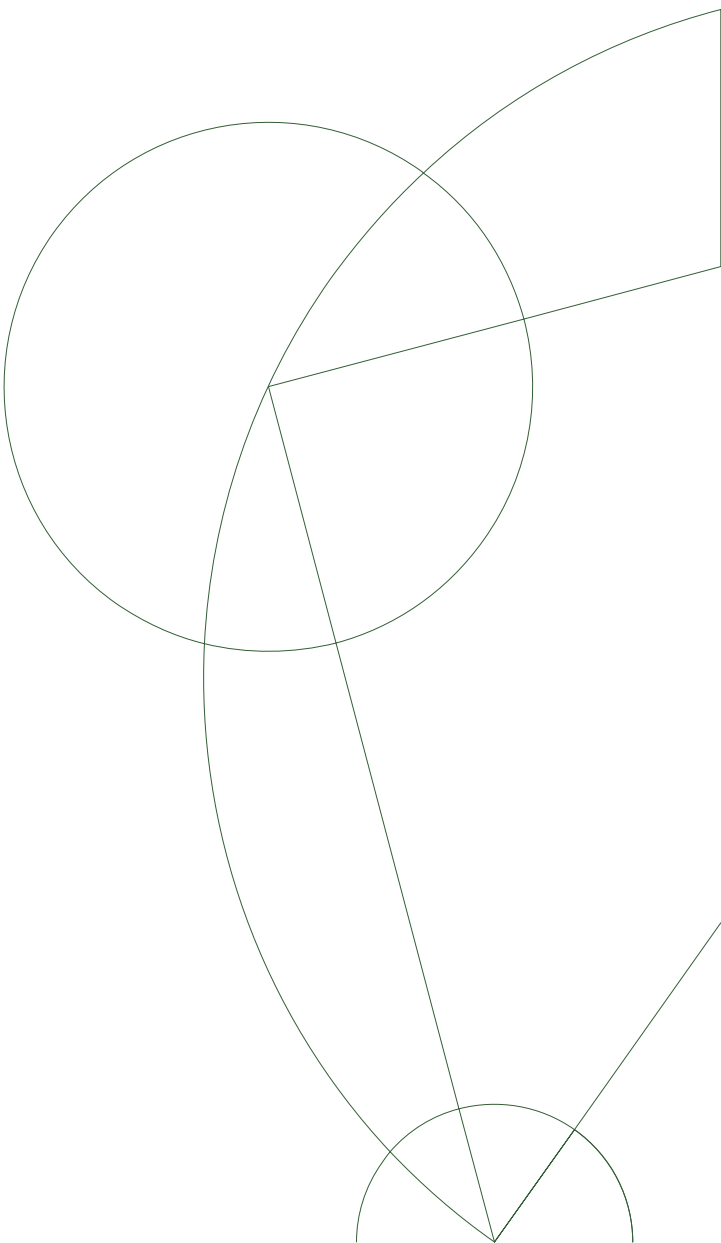
# G-Assignment: A Compiler for the Fasto Language

IPS 2018

Department of Computer Science
University of Copenhagen

Emil Masoumi<sjp318>, Tobias O. Stannius<qch898>
& Xueying Chen<fxg358>

14. March 2018



# Contents

# 1 Introduction

## 1.1 A FASTO Compiler

The present report details how we have implemented a compiler for the Fasto Language. Fasto is a strongly-typed, first-order-language, which supports arrays by special array constructors and combinators.

A compiler is a program that translates a source language into a target language, usually a lower level one; MIPS code in our case.

A compiler goes through a series of stages. The following phases are used in our design:

- **Lexing/Lexical analysis**
  The source code is first read and divided into tokens, each of which corresponds to a symbol in the programming language such as: a variable name, keyword or number.
- **Parsing/Syntax analysis**
  The list of tokens produced by the lexical analysis are arranged into a tree structure (an abstract syntax tree) that reflects the structure of the program.
- **Type checking**
  The abstract syntax tree is analyzed to determine if the program violates any typing rules, such as attempting to divide two booleans.
- **Register allocation**
  The symbolic variable names are translated to numbers, each of which corresponds to a register in the target machine code.
- **Machine code generation**
  The language is lastly translated into the target language. Our compiler translates it into MIPS.

In addition to the above, our implementation also includes an Interpreter and tools for optimizing the code, e.g. by copying constants and removing dead bindings.

## 1.2 Compiling and running the code

The `Fasto` library and tests may be compiled and tested via a UNIX-like command line in the following way:

1. Unzip `fasto.zip` and navigate to the `fasto` folder.
2. Compile the library by running this command:
   `make`
3. Run all tests using the supplied shell script, using this command:
   `./bin/runtests.sh`

# 2 Task 1 & 2 - Features

## 2.1 Changes to Lexer.fsl

In **Lexer.fs**, we add the following rules for tokens in order to recognise various operators, e.g. the multiplication and division operators:

```
rule Token = parse
    ...
    | '*'                   { Parser.MULT   (getPos lexbuf) }
    | '/'                   { Parser.DIV    (getPos lexbuf) }
    ...
```

Additionally, we extend the token rule that matches keywords with cases for boolean literals and function calls, e.g. true and replicate. Note that the boolean literal is passed a boolean value in addition to the position of the token, to assign it a value:

```
let keyword (s, pos) =
    match s with
        ...
        | "true"        -> Parser.BOOLLIT (true, pos)
        ...
        | "filter"      -> Parser.FILTER pos
        | "replicate"   -> Parser.REPLICATE pos
        | "scan"        -> Parser.SCAN pos
        ...
```

## 2.2 Changes to Parser.fsp

In **Parser.fs**, we have added various token definitions, e.g. the **MULT** and **BOOLLIT**-tokens, the first defined by a tuple of integers – the position of the token in the source text. The second has the position paired with a boolean value.
Immediately following these definitions, we define rules of association for the tokens to ensure unambiguous derivation. We place the association rule for **MULT** and **DIV** below e.g. **PLUS** and **MINUS**. The result of doing so is that the operators $*$ and / have precedence over $+$ and $-$.

```
\%token <bool*(int*int)> BOOLLIT
...
\%token <(int*int)> MULT DIV PLUS MINUS LESS AND OR NOT NEG
\%token <(int*int)> DEQ LTH EQ OP MAP REDUCE IOTA SCAN REPLICATE FILTER
     ARROW
...
\%left OR
\%left AND
\%right NOT
\%left DEQ LTH
\%left PLUS MINUS
\%left MULT DIV
\%left NEG
...
```

Lastly, we add grammar rules for the various expressions, e.g. for **DIV** and **SCAN**. For the first case, there must be an expression before and after the **DIV**-operator. This pattern is then translated to a **DIV**-token with the arguments $1

(first expression), $3 (second expression) and $2 (location of the token specified by line-number and column-number).
The **SCAN** pattern is slightly more complicated: it consists of the keyword, a left parenthesis, a function argument (FunArg), a comma, an expression (Exp), a comma, another expression and lastly a terminating right parenthesis. The corresponding token is passed the arguments $3 (function argument), $5 (first expression argument - should be an array), $7 (second expression argument - an accumulator of same type as contained in the array), unit (empty arg, which would otherwise be the type of the second expression) and lastly $1 – the position of the token in the code.

```
Exp : NUM            { Constant (IntVal (fst $1), snd $1) }
    ...
    | Exp DIV   Exp  { Divide($1, $3, $2) }
    ...
    | REPLICATE LPAR Exp COMMA Exp RPAR
                     { Replicate ($3, $5, (), $1) }
    | FILTER LPAR FunArg COMMA Exp RPAR
                     { Filter ($3, $5, (), $1) }
    | SCAN LPAR FunArg COMMA Exp COMMA Exp RPAR
                     { Scan ($3, $5, $7, (), $1) }
    ...
```

## 2.3 Changes to Interpreter.fs

Our changes to **Interpreter.fs** are solely to the **evalExp** function. It takes an untyped expression, a variable table and a function table, and recursively evaluates the expression via pattern matching, before returning the interpreted value or raising an error (typically type errors, since the program is untyped).
A simple pattern is that of the **DIVIDE** expression. The expressions contained are recursively evaluated with the vtab and ftab. The results are then pattern-matched to ensure type integrity. If both expressions result in **IntVal**s, division is performed and an **IntVal** is returned. Otherwise an exception is raised. Patterns such as **TIMES** function in a similar way.

```
| Divide(e1, e2, pos) ->
       let res1   = evalExp(e1, vtab, ftab)
       let res2   = evalExp(e2, vtab, ftab)
       match (res1, res2) with
         | (IntVal n1, IntVal n2) -> IntVal (n1/n2)
         | _ -> invalidOperands "Division on non-integral args: " [(
    Int, Int)] res1 res2 pos
```

Another simple pattern is **NEGATE**. As in the case above, the contained expression is recursively evaluated. If the result is an **IntVal**, we may negate it by subtracting it from 0 and returning the result as an **IntVal**. Alternatively we raise an error. **NOT** expressions are evaluated in a similar way, but for **BoolVal**s.

```
| Negate(e, pos) ->
    let res = evalExp(e, vtab, ftab)
    match res with
      | IntVal i -> IntVal (0-i)
      | _ -> raise (MyError("Negation expects an int value ", pos))
```

The interpreting of the expressions **AND** and **OR** is another matter entirely, since we want the Interpreter to have a short-circuit behaviour, i.e. for **AND**, the second expression need not be evaluated, if first expression is evaluated to true (for **OR**, the second expression is only evaluated, if the first returns false). As in the above cases, we raise an error, when attempting to evaluate invalid types.

```
| And (e1, e2, pos) ->
    let res1   = evalExp(e1, vtab, ftab)
    match res1 with
      | BoolVal true ->
        let res2 = evalExp(e2, vtab, ftab)
        match res2 with
          | BoolVal true  -> BoolVal true
          | BoolVal false -> BoolVal false
          | _ -> invalidOperand "&& on non-integral args: " Bool res2
    pos
      | BoolVal false -> BoolVal false
      | _ -> invalidOperand "&& on non-integral args: " Bool res1 pos
```

In addition to the aforementioned operators, we have also implemented a small range of functions in the Interpreter: **REPLICATE**, **FILTER** and **SCAN**.
First we will take a look at **REPLICATE** (code in appendix). We first recursively evaluate the *e* and *el* arguments, as well as the type of the *el* If *e* is indeed an **IntVal** and greater than or equal to 0, we use the **F# List.map**-function to copy *el* into a list of size *e*. If *e* is not a number or an invalid number, we raise an error.
Second, we will take a closer look at **FILTER**. We use the helper function rtpFunArg to lookup the predicate function 'p' in the function table. If the return type of the function is Bool, we continue to evaluate the array expression and if it is indeed an array consisting of a list of elements and their type, we use the **F#** List function **List.filter** to apply the 'p' function to the elements of the array, keeping only the elements that result in **BoolVal true**.

```
  | Filter (p, arrexp, t, pos) ->
        let farg_ret_type = rtpFunArg p ftab pos
        if (farg_ret_type = Bool) then
          let arr = evalExp(arrexp, vtab, ftab)
          match arr with
          | ArrayVal(lst, tp) ->
            let resVals = List.filter (fun a -> evalFunArg (p, vtab,
    ftab, pos, [a]) = BoolVal true) lst
            ArrayVal (resVals, tp)
          | _-> raise (MyError("Second Argument of Filter Is Not An
    Array: "+ppVal 0 arr, pos))
        else let msg = sprintf "Error: In Filter call, function return
    type is not a bool."
             raise (MyError(msg, pos))
```

Lastly, we will cover **SCAN** (code in appendix). For this case, we first recursively evaluate the array-expression and accumulator-expression. If *arrexp* is indeed an **ArrayVal**, we evaluate the function return type and check if it matches the array type (type checking the function and the accumulator is done elsewhere). If the aforementioned are equal, we match the length of the contained list with either 0 and return an empty **ArrayVal** *or* we initialize the accumulator *nel* in *first_el*, by calling **evalFunArg** on *farg* with *nel* and the head

of the list. We then use the **F#** List.scan function to evaluate the function *farg* with the accumulator *acc* and each remaining element *x* in the array (obtained using **List.tail**).

## 2.4 Changes in TypeChecker.fs

**Arithmetic operations**
All the four arithmetic operations are similar, hence if we take a starting point in Divide:

```
1     | Divide (e1, e2, pos) ->
2         let (t1, e1_dec) = checkExp ftab vtab e1
3         let (t2, e2_dec) = checkExp ftab vtab e2
4         if (t1 = t2 && t1 = Int)
5         then (Int, Divide (e1_dec, e2_dec, pos))
6         else raise (MyError ("In Divide: one of subexpression types is
    not Int: "+ppType t1+" and "+ppType t2, pos))
```

We see that we look up both expressions that are attempted to be divided, in the function table, and the variable table. The reason for this is because we want to determine their types. When looking them up, we get the result type tupled with the type-decorated expression. We then check if they have the same type, and they are both of type integer. If this is the case, we know that they consist of the correct type for conducting the division operation, and therefore, we recursively call *Divide* with type-decorated expression, and store it in *pos*. If it is not the case, we raise an error, because it is wrong to divide anything that are not of type integer.

**Logical Operations**
For our four logical operations, we have done something, somewhat similar as for our arithmetic operations. In *And* and *Or* we do the exact same thing, as for our arithmetic operations, but we recursively call the corresponding logical operation calls instead. Also the expressions looked up, must be booleans and not integers. However for *Not*, we only expect one expression, and expect it to be a boolean, therefore we return an error if it is not a boolean, and for *Negate*, we expect an integer to be negated, therefore we will return a type error if given something otherwise.

**Filter**
We have the following signature for filter:
*filter : ('T -> bool) -> 'T array -> 'T array*
where $'T$ is the same type. This means that filter takes a function that returns a boolean, and an array, and returns an array of the same type as the input array. The given function must also take the same type that the given array consists of. We first recursively check the the type of the second argument in the syntax tree. This will give us the type of the supposedly array and the type-decorated expression. After getting the type of the second argument, we then check if it is an array. We then check if the first argument is a function, and if it returns a boolean and its argument type is an array. If this is the case, we save the result in three corresponding variables and continue. If not, an error is returned.

```
let (f', f_res_type, f_arg_type) =
    match checkFunArg ftab vtab pos f with
      | (f', Bool, [a1]) -> (f', Bool, a1)
```

We then lastly check if the elements in the array is of equivalent type as the function argument. If that is the case we call *Filter* with the given function, type-decorated array, the array type and the position that we want to store it at.

**Scan**
Unlike our arithmetic and logical operations, the scan function should handle an array and return a new array according to the function and the accumulator it is given. First of all, we need to know that the input array is indeed an array; secondly, the given function is legit and has a consist element type; at last, the input/output array, the accumulator and the given function ought to have the same type/element type. Here we have used if-else statement and pattern matching to catch all possible types and raise errors if the checking types are not as expected. Code can be found in Appendix 6.4.1.

## 2.5 Code Generation

We now want to translate the logical and arithmetic operators that is defined for the FASTO language to machine code.

### 2.5.1 Arithmetic operations

**Multiplication and division:** Both these functions have been implemented in a similar way, hence if we take a starting point with multiplication:

```
  | Times (e1, e2, pos) ->
      let t1 = newName "multiply_L"
      let t2 = newName "multiply_R"
      let code1 = compileExp e1 vtable t1
      let code2 = compileExp e2 vtable t2
      code1 @ code2 @ [Mips.MUL (place,t1,t2)]
```

The pattern identifies three arguments: $e1$ and $e2$ are the type expressions, and *pos* is the location of the source code in the original text. We then begin with assigning our labels by using our name generator function that generate names for labels and temporary symbolic registers. We then recursively call the $compileExp$ function with our $e_n$ and $t_n$ to compile $e_n$, binding it to the variable table *vtable*, and placing the result in *place*. We then return with appending our compiled expression with the $MUL$ call for the MIPS instruction $MUL$ defined as *MUL($rd,$rs,$rt): $rd = $rs * $rt*. Note that addition, subtraction and division is implemented in exactly the same way, but their corresponding MIPS operations have been used in the place of the $MIPS.MUL$ operation.

### 2.5.2 Logical operators

We will walk through four logical operator in this section. First, we will have *Negate* and *Not* - they are similar to our arithmetic operators, the only differ-

ence is that they only have one type expression.
**Not:** Taking a look at *Not* we have:

```
| Not (e, pos) ->
    let t = newName "not"
    let code = compileExp e vtable t
    code @ [Mips.XORI(place, t, "1")]
```

We call the bitwise ORI operation, that takes the resulting register *place*, the source register *t* and an immediate 1. The result of this turns the leftmost bit in *t* to a 1, if it is a 0, and a 0 if it is a 1. Where 0 represents false and 1 represents true.
**Negate:** Similar to **Not** we have above, we have also only one type expression, yet it ought to execute twice bitwise ORI operation to achiever the negation:

```
| Negate (e, pos) ->
     let t = newName "negate"
     let code = compileExp e vtable t
     code @ [Mips.XORI(place, t, "1")] @ [Mips.XORI(place, t, "-1")]
```

Since XORI can be used to invert selected bits in a register, in order to handle both negative and positive digits, we let the source register *t* toggled by XORing with both 1 and -1.
As for **And/Or** these two functions, which are quite similar to each other, we will pick **And** as an example here:

```
| And (e1, e2, pos) ->
   let t1 = newName "and_L"
   let t2 = newName "and_R"
   let code1 = compileExp e1 vtable t1
   let code2 = compileExp e2 vtable t2
   let endLabel = newName "and_end"
   let falseLabel = newName "and_false"
   code1                                  // Compile exp1
   @ [Mips.BEQ (t1, "0", falseLabel)]     // Jump directly to false
   if exp1 false
   @ code2                                // Compile exp2
   @ [Mips.BEQ (t2, "0", falseLabel)]     // Jump to false if exp2
   false
   @ [Mips.LI  (place, "1")]              // Both true, load imm 1 =
   true
   @ [Mips.J endLabel]                    // Jump to end
   @ [Mips.LABEL falseLabel]              //
   @ [Mips.LI (place, "0")]               // One is false, load imm 0
    = false
   @ [Mips.LABEL endLabel]                // end
```

Since what we want to achieve is to fast forward the process, when the first expression meets the condition(false), we used labels for jumping. Likewise, **Or** function can also perform this trick, when the first expression is true. The process can be shorten, since the output will still be true logically.

### 2.5.3 Scan, Filter, Replicate

We want to translate some of the predefined functions from our FASTO language to machine code. The predefined functions that we seek to translate to machine code are: *replicate(n, a)*, *filter(f, arr)* and *scan(f, ne, arr)*.

### Scan

In order to fulfill scan's functionality, we have set up numbers of registers, e.g. length register that stores the size of the array, address register that stores the address of the input array, pointers, labels, etc.
A loop is needed when we want to handle an array. As for the loop structure, we have a counter register $i_{reg}$ and a $tmp_{reg}$ that stores the result of $i_{reg} - len_{reg}$, once $tmp_{reg} \leq 0$, the loop should terminated as this indicates there is the end of the input array.

```
| Scan (farg, acc_exp, arr_exp, elem_type, pos) ->
    let len_reg  = newName "len_reg"   (* size of input array *)
    let i_reg    = newName "ind_var"   (* loop counter *)
    let tmp_reg  = newName "tmp_reg"   (* for several purpose *)
    let loop_beg = newName "loop_beg"  (* start label for the loop *)
    let loop_end = newName "loop_end"  (* end label for the loop *)
    ...
    let init_regs = [ Mips.MOVE (i_reg, "0")
                    ; Mips.ADDI (x_it, arr_reg, "4")
                    ; Mips.ADDI (y_it, place, "4") ]

    let loop_header = [ Mips.LABEL(loop_beg)
                      ; Mips.SUB(tmp_reg, i_reg, len_reg)
                      ; Mips.BGEZ(tmp_reg, loop_end) ]
    ...
```

This piece of code above shows how registers are initialized and well-prepared, as well as how a loop is set. At this time, the element type of the input array should be took under consideration, since a boolean is 4-bytes long whereas a integer is 1-byte. That makes a difference for these two pointer registers: $x_{it}$ and $y_{it}$ - the first one points to wherever we are reading from the input array and the second one points to wherever we are writing onto the output array. When one element is read/written, the pointers should move to the next, according to the element type of the input array, it could either move 1 step or 4 steps forward. Therefore we have match the element size to solve this, which is shown below:

```
| Scan (farg, acc_exp, arr_exp, elem_type, pos) ->
    ...
    let x_it     = newName "x_it"      (* pointer for input array *)
    let y_it     = newName "y_it"      (* pointer for output array *)
    ...
    let loop_body =
        match elem_size with
        | One  -> [ Mips.LB (res_reg, x_it, "0") ]
                  @ applyFunArg (farg, [acc_reg; res_reg], vtable,
  acc_reg, pos) @
                  [ Mips.ADDI (x_it, x_it, "1")
                  ; Mips.SB (acc_reg, y_it, "0") ]
        | Four -> [ Mips.LW (res_reg, x_it, "0") ]
                  @ applyFunArg (farg, [acc_reg; res_reg], vtable,
  acc_reg, pos) @
                  [ Mips.ADDI (x_it, x_it, "4")
                  ; Mips.SW (acc_reg, y_it, "0") ]

    let loop_footer = [ Mips.ADDI (y_it, y_it, makeConst (
  elemSizeToInt(elem_size)))
                      ; Mips.ADDI (i_reg, i_reg, "1")
```

```
                        ; Mips.J loop_beg
                        ; Mips.LABEL loop_end ]
    ...
```

The code of scan in *CodeGen.fs* can be found in Appendix 6.5.1.

### Filter

We begin with allocating different temporary symbolic registers and labels for different causes. This is procedure that is first done by name generation when calling *NewName*. After this we initialize some of our registers in *init_regs*. Since *add_reg* is the address of the first element in the new array, we give it *place*, which is the address of the heap pointer, and then add it with 4 because all instructions are 32 bit. Since the $i$ and $j$ registers are our counters, we store 0 in them to begin with, and increment $i$ for each time we iterate through the input array, and increment $j$ for each time our function argument returns true. The register for the current element pointed at, gets the address of the array $+4$. All this is shown below:

```
let init_regs = [ Mips.ADDI (addr_reg, place, "4")
                ; Mips.MOVE (i_reg, "0")
                ; Mips.MOVE (j_reg, "0")
                ; Mips.ADDI (elem_reg, arr_reg, "4")
                ]
```

We then define the header of our loop, that is the beginning of our loop, which we want to revisit each time we are at the end of the body of our loop. In our loop header, we basically check if the $i$ register holds a value greater than or equal to the size of the array. If that is the case, we know that we are done, and branch to *loop_end*, else we continue in our function:

```
let loop_header = [ Mips.LABEL (loop_beg)
                  ; Mips.SUB (tmp_reg, i_reg, size_reg)
                  ; Mips.BGEZ (tmp_reg, loop_end)
                  ]
```

Now we first want to create our first result in the resulting array, and then iterate through the given array, applying the function and appending to the array from the right side. To do this we load whatever is in the element register into the resulting register, and call the given argument function for the current element stored in the element register. We then point to the next element by incrementing the address with one byte, if the type is a boolean or char, or with four bytes if the type is an int.

```
let loop_filter0 =
    match getElemSize elem_type with
        | One -> Mips.LB (res_reg, elem_reg, "0")
                 :: applyFunArg(farg, [res_reg], vtable, truefalse_reg,
     pos)
            @ [ Mips.ADDI (elem_reg, elem_reg, "1") ]
              | Four -> Mips.LW (res_reg, elem_reg, "0")
                        :: applyFunArg(farg, [res_reg], vtable,
    truefalse_reg, pos)
            @ [ Mips.ADDI (elem_reg, elem_reg, "4") ]
```

Now the area that the entire iteration happens is in the *loop_filter*1 assignment, where we branch if our argument function has returned false. What happens

in the branched label is that the $i$ register is incremented, because we want to process the next element in the array, so we jump to the initial label. If we however do not branch, we know that the argument function has returned true, and we want to increment both $i$ and $j$ registers, and store this element in the resulting register subsequently allocating a new address for the next element to come, because if we did not, we would simply just store every single element in the same address for our new array, resulting it in only consisting of 1 element. The intuition of $loop\_filter1$ is given below, and the entire code can be found in the appendix:

```
    | Four -> [ Mips.BEQ (truefalse_reg, "0", loop_footer)
              ; Mips.SW (res_reg, addr_reg, "0")
              ; Mips.ADDI (addr_reg, addr_reg, "4")
              ; Mips.ADDI (i_reg, i_reg, "1")
              ; Mips.ADDI (j_reg, j_reg, "1")
              ; Mips.J (loop_beg)
              ]
```

We obviously want to jump to the header of the loop after this.
The order that our functions are actually called in, and the way that they are actually stored, and the sequence in which they are called in, is defined within *arr_code*:

```
   arr_code
     @ get_size
     @ dynalloc (size_reg, place, elem_type)
     @ init_regs
     @ loop_header
     @ loop_filter0
     @ loop_filter1
     @ loop_footer
```

Replicate is implemented in a similar fashion.

# 3 Task 3 - Optimizations

## 3.1 CopyConstPropFold.fs

The changes made to this file enable us to perform a range of optimizations. We have added cases for the following expressions: **VAR**, **INDEX** and **LET (Dec (name, e, decpos), body, pos**, with various sub-cases. Additionally, we have extended **TIMES**, **PLUS** and **MINUS**
For the case of **VAR(v, pos)**, we look up the name of the variable in the variable symbol table. If a variable or constant propagate with that name already exists, the assignment is redundant, and we may replace it with the existing variable or constant. Otherwise we keep the assignment and name.
**INDEX** functions in a similar way, except it also recursively optimizes the expression used for indexing.
**LET (Dec (name, e, decpos), body, pos** has several sub-cases: **VAR**, **CONSTANT** and a nested **LET (Dec (name, e, decpos), body, pos**
In the sub-case for **VAR(v, pos)** we create a new variable propagate from v and bind it to the variable table, so that recursive calls may find the binding.

We then recursively optimize the body of the let-binding and return a new let-binding containing an optimized expression and optimized body (using the new propagate). The case for **CONSTANT** is handled in a similar way.
Lastly, we have implemented the sub-case for the nested let-binding, **Let (Dec (nested_name, nested_e, nested_decpos), nested_body, nested_pos)**. It restructures expressions of the form:

*let out_name = (let in_name = exp in in_body) in out_body*

to:

*let in_name = exp in let out_name = in_body in out_body*

and recursively evaluates the latter.
When an expression matches the nested let-binding, we define a *new* body with the outer name, the nested body as expression, followed by the outer body (line 6). Secondly, we define a normalized (restructured) expression with the inner name, inner expression, followed by the *new* body (line 7). Lastly we recursively call `copyConstPropFoldExp` on this normalized expression (line 8).

```
1 | Let (Dec (name, e, decpos), body, pos) ->
2     let e' = copyConstPropFoldExp vtable e
3     match e' with
4         ...
5         | Let (Dec (nested_name, nested_e, nested_decpos), nested_body,
       nested_pos) ->
6             let body' = Let (Dec (name, nested_body, decpos), body, pos
      )
7             let normalized_exp = Let (Dec (nested_name, nested_e,
      nested_decpos), body', pos)
8             copyConstPropFoldExp vtable normalized_exp
9         ...
```

Our changes to **TIMES** are minimal. We have added two additional cases (line 7-8) to the inner match of *(e1', e2')*. These allow us to optimize expressions such as $1 * Exp$ to just $Exp$, thus avoiding unnecessary computations. Note that we do not optimize the case of $0 * Exp$, since $Exp$ may contain IO-operations - we have chosen to only do safe optimizations. The changes to **PLUS** and **MINUS** are similar.

```
1 | Times (e1, e2, pos) ->
2     let e1' = copyConstPropFoldExp vtable e1
3     let e2' = copyConstPropFoldExp vtable e2
4     match (e1', e2') with
5         | (Constant (IntVal x, _), Constant (IntVal y, _)) ->
6             Constant (IntVal (x * y), pos)
7         | (Constant (IntVal 1, _), _) -> e2'
8         | (_, Constant (IntVal 1, _)) -> e1'
9         | _ -> Times (e1', e2', pos)
```

## 3.2 DeadBindingRemoval.fs

The functionality of this file is to remove the redundant code for the input code. Similar to what we have above in *CopyConstPropFold.fs*, it is also an optimization, that included many different cases, here we will walk through **VAR** and **LET (Dec (name, e, decpos), body, pos)** these two cases. Contain IO or not will make a significant difference in an express, thus we should handle them with

cautions.

Take a look at the sub-case for **VAR(name, pos)**, we can tell that there is not much for optimization. Since a variable name cannot contain IO, the tuple this case passes, has the first element: false. As for the rest, we have plainly bind the var name in an empty *SymTab* and pass the **Var (name, pos)** further.

In **LET (Dec (name, e, decpos), body, pos)**, we should split the whole expression into two parts: the head part and the body part. We defined *eios* and *bodyios* two booleans to indicate if the expression is IO-involved, meanwhile we could also know whether the var that binding to the body is in-used with a *uses* function. With these two considerations in mind, we used pattern matching to decide what what should be passed further. Only the body part will be passed down, when the let-binding var is not in-used. In the rest of cases, it will pass a boolean result of *eios* and *bodyios*, a binding *SymTab* and a typed expression.

The code of these two cases can be found in Appendix 6.6.

# 4 Testing

## 4.1 Features

Though the provided tests are rather comprehensive, we decided to write additional tests to ensure that our implementation works correctly. The screenshot below details the result of running all tests.

```
========== Running Fasto test programs ==========

Testing            booland.fo:  Success.
Testing            boollit.fo:  Success.
Testing            boolnot.fo:  Success.
Testing             boolor.fo:  Success.
Testing      comprehension.fo:  Parse error: Error at line 8, column 18
Compilation error.
Testing copyConstPropFold0.fo:  Success.
Testing copyConstPropFold1.fo:  Success.
Testing copyConstPropFold2.fo:  Success.
Testing       dead_bnd_rem.fo:  Success.
Testing               div.fo:  Success.
Testing         fail_parse.fo:  Success.
Testing               fib.fo:  Success.
Testing    filter-on-2darr.fo:  Success.
Testing         filter-two.fo:  Success.
Testing            filter.fo:  Success.
Testing        filter_bool.fo:  Success.
Testing         inline_map.fo:  Success.
Testing            io_mssp.fo:  Success.
Testing              iota.fo:  Success.
Testing            lambda.fo:  Success.
Testing         map_red_io.fo:  Success.
Testing              mult.fo:  Success.
Testing         neg_simple.fo:  Success.
Testing            negate.fo:  Success.
Testing            ordchr.fo:  Success.
Testing       proj_figure3.fo:  Success.
Testing            reduce.fo:  Success.
Testing          replicate.fo:  Success.
Testing              scan.fo:  Success.
Testing          scan_bool.fo:  Success.
Testing      short_circuit.fo:  Success.
```

**Figure 1:** Result of running the runtests.sh script.

As is evident from the illustration, all tests but comprehension.fo pass, since we have not implemented this feature. Below are brief descriptions of selected tests. The remaining are described in the appendix.

- booland.fo and boolor.fo tests the **&&** and **||**-operators and the short-circuit implementations separately.
- boollit.fo tests the boolean literals; **true** and **false**.
- boolnot.fo tests the **not**-operator
- div.fo simply tests the **/**-operator
- fail_parse.fo tests that the compiler raises an appropriate exception.
- fib.fo tests various operators and elements in unison: **+**, **-** and **==**, as well as **if-then-else** (with a nested **if-then-else**).
- filter-two.fo also tests a combination of functions: lambda-functions, **filter**, **iota** and **map**.
- filter.fo and filter_bool.fo also tests various functions, but especially **filter** with different types: **int** and **bool**.

- inline_map.fo has a declared purpose of testing the inline-placement of the map function, but may also serve to test our optimizations.
- iota.fo tests the **iota**-function.
- lambda.fo tests lambda-functions / operators passed as arguments.
- mult.fo tests the * operator.
- neg_simple.fo tests that the compiler raises a type error.
- negate.fo tests the ~-operator.
- reduce.fo tests the **reduce**-function.
- replicate.fo tests the **replicate**-function
- scan.fo and scan_bool.fo tests the **scan**-function with various types: **int** and **bool**.
- short_circuit.fo also tests the short-circuit implementaiton of **&&** and **||**.

## 4.2 Optimizations

Testing whether our optimizations work correctly requires a more qualitative approach. Optimization tests include the following:

- copyConstPropFold0.fo
- copyConstPropFold1.fo
- copyConstPropFold2.fo
- dead_bnd_rem.fo

Let us consider the dead_bnd_rem2.fo-test. Below is the input and result of optimizing the code with the pipeline: *icdcdD*

```
// Input
fun int f(int x, int y) =
  (x + 2) * (y - 2)

fun int main() =
  let a = read(int) in
  let b = let x = a in
          let y = 2 in
          (x + 2) * (y - 2)
  in b

// Output
fun int main() =
    let a = read(int) in
    let b = ((a + 2) * 0) in
    b
```

Our optimizations first perform copy-constant propagation, replacing $x$ with $a$ and $y$ with 2, which will later be used to optimize $(y - 2)$ to 0. Lastly we remove the dead bindings x = a and y = 2 and obtain the output shown above.

# 5 Conclusion

We have managed to implement all to-do task, besides the bonus one. The code compiles, all tests are passed successfully and the optimization looks good.

# 6 Appendix

## 6.1 Overview of implemented features and optimizations

| | Interpreting | Type Checking | Code Generation |
|---|---|---|---|
| Times | x | x | x |
| Divide | x | x | x |
| Negate | x | x | x |
| And | x | x | x |
| Or | x | x | x |
| Not | x | x | x |
| True | x | - | x |
| False | x | - | x |
| Replicate | x | x | x |
| Filter | x | x | x |
| Scan | x | x | x |

**Table 1:** Implemented features

| | CopyConstPropFold | removeDeadBindingsinExp |
|---|---|---|
| Var | x | x |
| Index | x | x |
| Let | x | x |
| Let Var | x | - |
| Let Constant | x | - |
| Let Let | x | - |
| Times | x | - |
| Divide | x | - |
| Plus | - | - |
| Minus | - | - |
| And | x | - |
| Or | - | - |
| Equal | - | - |
| Not | - | - |

**Table 2:** Implemented optimizations

## 6.2 Interpreter.fs

### 6.2.1 replicate

```
| Replicate (e, el, t, pos) -> // e: exp size, el: exp element to repeat, t: type of el, pos: position
    let n = evalExp(e, vtab, ftab)
    let a = evalExp(el, vtab, ftab)
    let tp = valueType a
    match n with
      | IntVal size ->
        if size >= 0
        then ArrayVal( List.map (fun x -> a) [0..size-1], tp )
        else let msg = sprintf "Error: In Replicate call, size is negative: %i" size
             raise (MyError(msg, pos))
      | _ -> raise (MyError("Replicate argument is not a number: "+ppVal 0 n, pos))
```

### 6.2.2 scan

```
| Scan (farg, ne, arrexp, tp, pos) ->
    let farg_ret_type = rtpFunArg farg ftab pos
    let arr = evalExp (arrexp, vtab, ftab)
    let nel = evalExp (ne, vtab, ftab)
    match arr with
      | ArrayVal (lst, tp1) ->
         match lst.Length with
           | 0 -> ArrayVal([], tp1)
           | _ -> let first_el = (evalFunArg(farg, vtab, ftab, pos, [nel;List.head(lst)]))
                  ArrayVal (List.scan (fun acc x -> evalFunArg (farg, vtab, ftab, pos, [acc;x]))
                                      first_el (List.tail(lst)), tp1)
      | other -> raise (MyError ("Third argument of scan is not an array :"
                                 + ppVal 0 arr, pos))
```

### 6.2.3 filter

```
| Filter (p, arrexp, t, pos) ->
    let farg_ret_type = rtpFunArg p ftab pos
    if (farg_ret_type = Bool) then
      let arr = evalExp(arrexp, vtab, ftab)
      match arr with
          | ArrayVal(lst, tp) ->
            let resVals = List.filter (fun a -> evalFunArg (p, vtab, ftab, pos, [a]) = BoolVal true) lst
            ArrayVal (resVals, tp)
          | _-> raise (MyError("Second Argument of Filter Is Not An Array: "+ppVal 0 arr, pos))
        else let msg = sprintf "Error: In Filter call, function return type is not a bool."
             raise (MyError(msg, pos))
```

## 6.3 TypeChecker.fs

### 6.3.1 scan

```
 | Scan (f, n_exp, arr_exp, _, pos) ->
        let (n_type  , n_dec  ) = checkExp ftab vtab n_exp
        let (arr_type, arr_dec) = checkExp ftab vtab arr_exp
        let elem_type =
            match arr_type with
```

```
            | Array t -> t
            | other -> raise (MyError ("Scan: Argument not an array",
 pos))
      let (f', f_arg_type) =
          match checkFunArg ftab vtab pos f with
            | (f', res, [a1; a2]) ->
                if a1 = a2 && a2 = res
                then (f', res)
                else raise (MyError( "Scan: incompatible function
type of " +
                                     (ppFunArg 0 f) + ": " +
showFunType ([a1; a2], res)
                                   , pos))
            | (_, res, args) ->
                raise (MyError ( "Scan: incompatible function type of
 " +
                                 ppFunArg 0 f + ": " + showFunType (
args, res)
                                , pos))
      let err (s, t) = MyError ( "Scan: unexpected " + s + " type " +
 ppType t +
                                 ", expected " + ppType f_arg_type
                                , pos)
      if   elem_type = f_arg_type && elem_type = n_type then
           (arr_type, Scan (f', n_dec, arr_dec, elem_type, pos))
      elif elem_type = f_arg_type then
           raise (err ("neutral element", n_type))
      else raise (err ("array element", elem_type))
```

```
  | 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
            | other   -> raise (MyError ("Filter: Argument not an
array", pos))
      let (f', f_res_type, f_arg_type) =
          match checkFunArg ftab vtab pos f with
            | (f', Bool, [a1]) -> (f', Bool, a1)
            | (_,  _, args) ->
                raise (MyError ( "Filter: incompatible function type
of " +
                                  ppFunArg 0 f + ":" + showFunType (
args, Bool)
                                , pos ))
      if elem_type = f_arg_type
      then ( Array f_arg_type
           , Filter (f', arr_exp_dec, elem_type, pos) )
      else raise (MyError( "Filter: array element types does not
match." +
                           ppType elem_type + " instead of " +
ppType f_arg_type
                         , pos))
```

## 6.4 CodeGen.fs

### 6.4.1 scan

```
| Scan (farg, acc_exp, arr_exp, elem_type, pos) ->
    let len_reg  = newName "len_reg"    (* size of input array *)
```

```
3       let arr_reg  = newName "arr_reg"   (* address of array *)
4       let i_reg    = newName "ind_var"   (* loop counter *)
5       let acc_reg  = newName "acc_reg"   (* accelerator *)
6       let tmp_reg  = newName "tmp_reg"   (* for several purpose *)
7       let res_reg  = newName "res_reg"   (* for several purpose *)
8       let loop_beg = newName "loop_beg"  (* start label for the loop *)
9       let loop_end = newName "loop_end"  (* end label for the loop *)
10      let x_it     = newName "x_it"      (* pointer for input array *)
11      let y_it     = newName "y_it"      (* pointer for output array *)
12
13      let elem_size = getElemSize elem_type
14      let arr_code  = compileExp arr_exp vtable arr_reg
15      let acc_code  = compileExp acc_exp vtable acc_reg
16      let len_code  = [ Mips.LW (len_reg, arr_reg, "0") ]
17
18      let init_regs = [ Mips.MOVE (i_reg, "0")
19                      ; Mips.ADDI (x_it, arr_reg, "4")
20                      ; Mips.ADDI (y_it, place, "4") ]
21
22      let loop_header = [ Mips.LABEL(loop_beg)
23                        ; Mips.SUB(tmp_reg, i_reg, len_reg)
24                        ; Mips.BGEZ(tmp_reg, loop_end) ]
25
26      let loop_body =
27          match elem_size with
28          | One  -> [ Mips.LB (res_reg, x_it, "0") ]
29                    @ applyFunArg (farg, [acc_reg; res_reg], vtable,
    acc_reg, pos) @
30                    [ Mips.ADDI (x_it, x_it, "1")
31                    ; Mips.SB (acc_reg, y_it, "0") ]
32          | Four -> [ Mips.LW (res_reg, x_it, "0") ]
33                    @ applyFunArg (farg, [acc_reg; res_reg], vtable,
    acc_reg, pos) @
34                    [ Mips.ADDI (x_it, x_it, "4")
35                    ; Mips.SW (acc_reg, y_it, "0") ]
36
37      let loop_footer = [ Mips.ADDI (y_it, y_it, makeConst (
    elemSizeToInt(elem_size)))
38                        ; Mips.ADDI (i_reg, i_reg, "1")
39                        ; Mips.J loop_beg
40                        ; Mips.LABEL loop_end ]
41
42      arr_code
43        @ acc_code
44        @ len_code
45        @ dynalloc (len_reg, place, elem_type)
46        @ init_regs
47        @ loop_header
48        @ loop_body
49        @ loop_footer
```

### 6.4.2 filter

```
1   | Filter (farg, arr_exp, elem_type, pos) ->
2       let size_reg = newName "size_reg" (* size of input/output array
    *)
3       let arr_reg  = newName "arr_reg"  (* address of array *)
4       let elem_reg = newName "elem_reg" (* address of single element *)
5       let res_reg  = newName "res_reg"  (* resulting array *)
6       let tmp_reg = newName "tmp_reg"
```

```
7       let truefalse_reg = newName "truefalse_reg" (* result of f(a) *)
8       let arr_code = compileExp arr_exp vtable arr_reg
9
10      let get_size = [ Mips.LW (size_reg, arr_reg, "0") ]
11
12      let addr_reg = newName "addr_reg" (* address of element in new
    array *)
13      let j_reg = newName "j_reg" (*Amount of true's returned by the
    function*)
14      let i_reg = newName "i_reg"
15
16      let init_regs = [ Mips.ADDI (addr_reg, place, "4")
17                      ; Mips.MOVE (i_reg, "0")
18                      ; Mips.MOVE (j_reg, "0")
19                      ; Mips.ADDI (elem_reg, arr_reg, "4")
20                      ]
21
22      let loop_beg = newName "loop_beg"
23      let loop_end = newName "loop_end"
24      let loop_footer = newName "loop_footer"
25
26      let loop_header = [ Mips.LABEL (loop_beg)
27                        ; Mips.SUB (tmp_reg, i_reg, size_reg) //tmp_reg
     = i_reg - size_reg
28                        ; Mips.BGEZ (tmp_reg, loop_end) //if tmp_reg >=
     0 goto loop_end
29                        ]
30
31      (* filter is 'arr[i] = (f(old_arr[i]) => true)' *)
32      let loop_filter0 =
33          match getElemSize elem_type with
34              | One -> Mips.LB (res_reg, elem_reg, "0")
35                            :: applyFunArg(farg, [res_reg], vtable,
    truefalse_reg, pos)
36                           @ [ Mips.ADDI (elem_reg, elem_reg, "1") ]
37              | Four -> Mips.LW (res_reg, elem_reg, "0")
38                            :: applyFunArg(farg, [res_reg], vtable,
    truefalse_reg, pos)
39                            @ [ Mips.ADDI (elem_reg, elem_reg, "4") ]
40
41      let loop_filter1 =
42          match getElemSize elem_type with
43              | One  -> [ Mips.BEQ (truefalse_reg, "0", loop_footer) //
    if (f(a)){GOTO loop_footer}
44                        ; Mips.SB (res_reg, addr_reg, "0")
45                        ; Mips.ADDI (addr_reg, addr_reg, "1")
46                        ; Mips.ADDI (i_reg, i_reg, "1")
47                        ; Mips.ADDI (j_reg, j_reg, "1")
48                        ; Mips.J (loop_beg)
49                        ]
50              | Four -> [ Mips.BEQ (truefalse_reg, "0", loop_footer) //
    if (f(a)){GOTO loop_footer}
51                        ; Mips.SW (res_reg, addr_reg, "0")
52                        ; Mips.ADDI (addr_reg, addr_reg, "4")
53                        ; Mips.ADDI (i_reg, i_reg, "1")
54                        ; Mips.ADDI (j_reg, j_reg, "1")
55                        ; Mips.J (loop_beg)
56                        ]
57
58      let loop_footer =
```

```
          [ Mips.LABEL (loop_footer)
          ; Mips.ADDI (i_reg, i_reg, "1")
          ; Mips.J (loop_beg)
          ; Mips.LABEL (loop_end)
          ; Mips.SW (j_reg, place, "0")
          ]

      arr_code
        @ get_size
        @ dynalloc (size_reg, place, elem_type)
        @ init_regs
        @ loop_header
        @ loop_filter0
        @ loop_filter1
        @ loop_footer
```

### 6.4.3 replicate

```
| Replicate (n_exp, a_exp, tp, pos) ->
    let size_reg = newName "size_reg"
    let n_code = compileExp n_exp vtable size_reg

    (* Check that array size N >= 0:
        if N - 1 >= 0 then jumpto safe_lab
        jumpto "_IllegalArrSizeError_"
        safe_lab: ...
    *)

    let safe_lab = newName "safe_lab"
    let checksize = [ Mips.ADDI (size_reg, size_reg, "-1")
                    ; Mips.BGEZ (size_reg, safe_lab)
                    ; Mips.LI ("5", makeConst (fst pos))
                    ; Mips.J "_IllegalArrSizeError_"
                    ; Mips.LABEL (safe_lab)
                    ; Mips.ADDI (size_reg, size_reg, "1")
                    ]

    (* init counter i and array address in register *)
    let addr_reg = newName "addr_reg"
    let i_reg = newName "i_reg"
    let a_reg = newName "a_reg"
    let a_code = compileExp a_exp vtable a_reg
    let init_regs = [ Mips.ADDI (addr_reg, place, "4")
                    ; Mips.MOVE (i_reg, "0") ]
                   @ a_code
    (* addr_reg is now the position of the first array element. *)

    (* Enter loop. Check if i_reg < size_reg. If true, continue
        If false, jump to loop_end. *)
    let loop_beg = newName "loop_beg"
    let loop_end = newName "loop_end"
    let tmp_reg = newName "tmp_reg"
    let loop_header = [ Mips.LABEL (loop_beg)
                      ; Mips.SUB (tmp_reg, i_reg, size_reg)
                      ; Mips.BGEZ (tmp_reg, loop_end)
                      ]
    let loop_replicate =
        match getElemSize tp with
            | One  -> [ Mips.SB (a_reg, addr_reg, "0") ]
            | Four -> [ Mips.SW (a_reg, addr_reg, "0") ]
```

```
43
44      let loop_footer = [ Mips.ADDI(addr_reg, addr_reg,
45                                    makeConst (elemSizeToInt (
    getElemSize tp)))
46                        ; Mips.ADDI (i_reg, i_reg, "1")
47                        ; Mips.J loop_beg
48                        ; Mips.LABEL loop_end
49                        ]
50
51      n_code
52      @ checksize
53      @ dynalloc (size_reg, place, Int)
54      @ init_regs
55      @ loop_header
56      @ loop_replicate
57      @ loop_footer
```

## 6.5 CopyConstPropFold.fs

### 6.5.1 Var

```
1  | Var (name, pos) ->
2          let id = SymTab.lookup name vtable
3          match id with
4              | Some (VarProp v)   -> Var(v, pos)
5              | Some (ConstProp c) -> Constant(c, pos)
6              | _                  -> Var(name, pos)
```

### 6.5.2 Index

```
1  | Index (name, e, t, pos) ->
2          let id = SymTab.lookup name vtable
3          let e' = copyConstPropFoldExp vtable e
4          match id with
5              | Some (VarProp v)   -> Index(v, e', t, pos)
6              | _                  -> Index(name, e', t, pos)
```

## 6.6 DeadBindingRemoval.fs

### 6.6.1 Var

```
1   | Var (name, pos) ->
2              let new_tab = SymTab.empty()
3              (
4                  false,
5                  SymTab.bind name () new_tab,
6                  Var (name, pos)
7              )
```

### 6.6.2 Let (Dec (name, e, decpos), body, pos)

```
1   | Let (Dec (name, e, decpos), body, pos) ->
2              let (eios, e_uses, e') = removeDeadBindingsInExp e
3              let (bodyios, body_uses, body') = removeDeadBindingsInExp
    body
4              let uses = SymTab.lookup name body_uses
5              match (uses, eios) with
6                  | (None, false) -> (bodyios, body_uses, body')
7                  | (_, _) -> (eios || bodyios, SymTab.combine e_uses
    body_uses, Let (Dec (name, e', decpos), body', pos))
```

## 6.7 Testing

Description of tests not covered above:

- filter-on-2darr.fo tests a combination of functions: lambda-functions, **filter**, **map** and **reduce**.
- io_mssp.fo computes the **maximal segment sum** using various operators and functions.
- map_red_io.fo tests map, reduce and various operators.
- ordchr.fo tests **map** with some built-in functions.
- proj_figure3.fo tests computations on arrays.