# G-Assignment

Implementation of Programming Languages (IPS) 2021

Department of Computer Science
University of Copenhagen

Tobias Bonnesen <cbv362@alumni.ku.dk>
Thor V.A.N. Olesen <mfz360@alumni.ku.dk>
Joachim Fiil <gmb114@alumni.ku.dk>

June 1, 2021

## Contents



## Introduction

This report outlines the completion of the partial implementation to the Fasto compiler written in F# as host language and MIPS as target language. This includes describing all compiler phases with a particular focus on lexing, parsing, interpretation, type checking, interpretation, machine code generation, and optimizations. A section on intermediate code generation has been omitted, since Fasto expressions (i.e., abstract syntax tree) are translated directly to MIPS machine code in our host language, F#.

# Implementation

## Feature 1: Arithmetic and Boolean Expressions

### Integer Multiplication and Division

Implementing the productions

$$Exp \to Exp * Exp$$
$$Exp \to Exp \ / \ Exp$$

`Lexer`

```
  | '*'                 { Parser.TIMES (getPos lexbuf) }
  | '/'                 { Parser.DIVIDE (getPos lexbuf) }
```

`Parser`

```
%token <Position> PLUS MINUS LESS TIMES DIVIDE
```

See other code snippet for precedence and associativity. The added grammar can be seen below.

```
      | Exp TIMES Exp  { Times($1, $3, $2) }
      | Exp DIVIDE Exp { Divide($1, $3, $2) }
```

`Type Checker`

```
      | Times (e1, e2, pos) ->
          let (e1_decorator, e2_decorator) = checkBinOp ftab vtab (pos,
          ↪  Int, e1, e2)
          (Int, Times (e1_decorator, e2_decorator, pos))

      | Divide (e1, e2, pos) ->
          let (e1_decorator, e2_decorator) = checkBinOp ftab vtab (pos,
          ↪  Int, e1, e2)
          (Int, Divide (e1_decorator, e2_decorator, pos))
```

`Interpreter`

```
    | Times (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 "Times on non-integral args: " [ (Int, Int) ] res1 res2 pos
    | Divide (e1, e2, pos) ->
        let res1 = evalExp (e1, vtab, ftab)
        let res2 = evalExp (e2, vtab, ftab)
```

`Code Generator`

Multiplication and division is implemented similarly to addition and subtraction, using corresponding Mips instructions. In addition, the division operation needs to catch attempts to divide by zero. So far, we nicely handle compile time error, but compiled Fasto programs attempting to divide by zero will not fail gracefully.

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

  | Divide (e1, e2, (line, _)) ->
     let t1 = newReg "divide_L"
     let t2 = newReg "divide_R"
     let code1 = compileExp e1 vtable t1
     let code2 = compileExp e2 vtable t2
     let safe_label = newLab "safe_lab"
     let checkdivzero = [ Mips.BNE (t2, RZ, safe_label) // if t2 is not 0, then jump to safe label, otherwise continue
                        ; Mips.LI (RN5, line) // load error line into register 5
                        ; Mips.LA (RN6, "_Msg_DivZero_") // load address of div zero error message into register 6
                        ; Mips.J "_RuntimeError_" // jump to runtime error code
                        ; Mips.LABEL (safe_label) ]
     code1 @ code2 @ checkdivzero @ [Mips.DIV (place,t1,t2)]
```

### Boolean Literals: `true` and `false`

We are implementing the productions

$$Exp \rightarrow \texttt{true}$$
$$Exp \rightarrow \texttt{false}$$

#### `Lexer`

In `Lexer.fs` we add “`true`” and “`false`” to the list of keywords, which is then mapped to a `BOOLLIT` token with corresponding boolean value and position in `Parser.fsp`. The relevant code in `Lexer.fsl` is shown below.

```
        | "true"        -> Parser.BOOLLIT (true, pos)
        | "false"       -> Parser.BOOLLIT (false, pos)
```

#### `Parser`

See description above. The relevant code in `Parser.fsp` is show below.

```
%token <bool * Position> BOOLLIT
```

#### `Type Checker`

Type checking of the boolean literals is handled in `TypeChecker.fs` by the already implemented decoration of `CONSTANT` types seen below.

```
      | Constant  (v, pos) -> (valueType v, Constant (v, pos))
```

#### `Interpreter`

The interpretation is also already handled in `Interpreter.fs`:

```
      | Constant (v, _) -> v
```

#### `Code Generator`

Code generation is handled by representing `true` and `false` as 1 and 0, putting the corresponding value in the `place` register using a MIPS *load-immediate* instruction as shown below.

```
  | Constant (BoolVal p, _) ->
      if p then
        [ Mips.LI (place, 1) ]
      else
        [ Mips.LI (place, 0) ]
```

### Boolean Binary Operators: `&&` and `||`

Implementing the productions

$$Exp \to Exp \texttt{ \&\& } Exp$$
$$Exp \to Exp \texttt{ || } Exp$$

#### `Lexer`

We add “`&&`” and “`||`” as individual tokens with corresponding positions. The relevant code in `Lexer.fsl` is shown below.

```
  | "&&"                    { Parser.AND     (getPos lexbuf) }
  | "||"                    { Parser.OR      (getPos lexbuf) }
```

#### `Parser`

See description of token above. The relevant code in `Parser.fsp` is shown below.

```
%token <Position> AND OR
```

Operator precedence and associativity is implemented in `Parser.fsp`.

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

Below is shown the added grammar in `Parser.fsp`.

```
    | Exp AND   Exp  { And ($1, $3, $2)  }
    | Exp OR    Exp  { Or  ($1, $3, $2)  }
```

### Type Checker

Type checking is handled in a similar way as the binary operators for integer arithmetics are handled. The relevant code in `TypeChecker.fs` is shown below.

```
    | And (e1, e2, pos) ->
        let (e1_decorator, e2_decorator) = checkBinOp ftab vtab (pos, Bool, e1, e2)
        (Bool, And (e1_decorator, e2_decorator, pos))

    | Or (e1, e2, pos) ->
        let (e1_decorator, e2_decorator) = checkBinOp ftab vtab (pos, Bool, e1, e2)
        (Bool, Or (e1_decorator, e2_decorator, pos))
```

### Interpreter

The interpretation of `&&` and `||` is implemented with *short-circuiting*, i.e., for the `&&` operation, the second operand is only evaluated if the first operand evaluates to `true`; and for the `||` operator, the second operand is only evaluated if the first operand evaluates to `false`. The implementation in `Interpreter.fs` is shown below.

```
    | And (e1, e2, pos) ->
       let res1 = evalExp (e1, vtab, ftab)

       match res1 with
        | BoolVal false -> BoolVal false
        | BoolVal true  ->
            let res2 = evalExp (e2, vtab, ftab)
            match res2 with
            | BoolVal b2 -> BoolVal b2
            | _ ->  invalidOperand "And on non-boolean arg: " Bool res2 pos
        | _ ->  invalidOperand "And on non-boolean arg: " Bool res1 pos

    | Or (e1, e2, pos) ->
       let res1 = evalExp (e1, vtab, ftab)

       match res1 with
        | BoolVal true  -> BoolVal true
        | BoolVal false ->
            let res2 = evalExp (e2, vtab, ftab)
            match res2 with
            | BoolVal b2 -> BoolVal b2
            | _ -> invalidOperand "And on non-boolean arg: " Bool res2 pos
        | _ -> invalidOperand "And on non-boolean arg: " Bool res1 pos
```

`Code Generator`

Code generation for `&&` and `||` has been implemented with *short-circuiting*. In the generated sequence of MIPS instructions, the instructions for evaluating the expression on the righthand side are only executed if strictly necessary to determine the value of the conjunction/disjunction.

```
      | And (c1, c2, pos) ->
        let t1 = newReg "and_L"
        let t2 = newReg "and_R"
        let code1 = compileExp c1 vtable t1
        let code2 = compileExp c2 vtable t2
        let falseLabel = newLab "falseLabel"
        let shortcircuit = [ Mips.LI (place, 0) ; Mips.BEQ (t1, RZ,
        ↪  falseLabel) ]
        code1 @ shortcircuit @ code2 @ [Mips.AND (place, t1, t2);
        ↪  Mips.LABEL falseLabel]

    | Or (c1, c2, pos) ->
        let t1 = newReg "and_L"
        let t2 = newReg "and_R"
        let code1 = compileExp c1 vtable t1
        let code2 = compileExp c2 vtable t2
        let trueLabel = newLab "trueLabel"
        let shortcircuit = [ Mips.LI (place, 1) ; Mips.BNE (t1, RZ,
        ↪  trueLabel) ]
        code1 @ shortcircuit @ code2 @ [Mips.OR (place, t1, t2); Mips.LABEL
        ↪  trueLabel]
```

### Boolean Negation: `not`

`Code Generator`

Code generation for the `not` operation is handled by first reading the value of the boolean operand expression into the `place` register and putting the result of an `XORI` MIPS instruction on the destination `place` register and an immediate value into the `place` register. Put simply, bitwise negation can be done by XORing with the immediate value 1 so the XOR instruction will invert every bit in the destination place register, which holds the boolean value that is then effectively negated.

```
    | Not (e, pos) ->
         // read value of e into register 'place'
         // then negate using XORI
         let code = compileExp e vtable place
         code @ [ Mips.XORI (place, place, 1) ]
```

### Arithmetic Negation: ~

`Code Generator`

```
  | Negate (e, pos) ->
      let code = compileExp e vtable place
      code @ [ Mips.SUB (place, RZ, place) ]
```

## Feature 2: Array Combinators

All array combinators are trivially added in Lexer.fsl, as keywords to be matched with in the keyword function:

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

In Parser.fsp, tokens are added for all of them:

```
%token <Position> REPLICATE FILTER SCAN
```

And grammar rules are added. The tokens to be matched for each expression follows naturally from how we would write the function call in Fasto:

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

We know how to unpack the values from the tokens to the Array Combinator Datatype constructors, by looking up each of the datatype constructors in AbSyn.fs. E.g. we can lookup Filter to be on the form:

```
Filter of FunArg<'T> * Exp<'T> * 'T * Position
```

Since *FunArg* is the *third* token in the Filter-expression match, we give it as the first argument to the Filter datatype constructor, by use of *\$3*.

All the datatype constructors take a generic type parameter *'T* as argument. Since we don't know the type yet, we pass "()" as the type to the constructors.

### Interpreter

For the Interpreter part, we leverage the fact that F# has built-in functions that corresponds to our array constructors. I.e. for the Fasto `replicate` construct, we use the F# `List.replicate` and for the second order array combinators (SOACs) `filter` and `scan`, we use F#'s `List.filter` and `List.scan`. For all of them, we wrap the resulting F# list in the AbSyn ArrayVal type.

For `Replicate (e1, e2, t, pos)`, we evaluate expressions e1 and e2, and by matching on the result of e1, we ensure that if e1 is not greater or equal to 0, we raise an error.

For `Filter (farg, arrexp, _, pos)`, we use the function `rtpFunArg` to get the return type of `farg` and raise an error if the return type is not Bool. We also evaluate `arrexp`, which should result in an ArrayVal. We match on it, and if it's not an ArrayVal, we raise an error, explaining that it's not an array. Otherwise, we use `List.filter` to generate the resulting array and wrap it in a ArrayVal.

### MIPS Machine Code Generation

#### `Replicate`

The code generation of MIPS instructions for the `replicate(n,a)` array combinator function checks that the size of the array is nonnegative before allocating memory for the new array using the provided `dynalloc` function with corresponding size of elements. Skipping the first four bytes of the allocated memory, the input element is copied to the array using a for-loop. If the size of array is negative, the program terminates with an error, which is handled similarly to that of the `iota` function.

#### `Filter`

The code generation for the `filter` array combinator is very similar to that of `map`. However, the element size remains the same for the resulting array and the number of elements in the resulting array is at most the number of elements in the input array. Loop over the elements of the input array, and with a seperate counter for the resulting array, the `res_reg` symbolic register is used as a sort of multi-purpose register: it is used to store the argument for the function `f`, and to store the boolean result of the function as well, which is then used to determine whether the corresponding element from the input array should be copied to the result array or not. Finally, the first word of the resulting array is updated with the correct length of the array. Note, that we do not handle updating the heap pointer correspondingly.

`Scan`

The code generation for the scan array combinator is inspired heavily from the reduce array combinator. The only difference is that we maintain an iterator (i.e., accumulator) to keep track of the last computed result between each iteration through the array, which becomes the result of the output array. The code generation extracts the size of the input array using a load word instruction (i.e., 4 byte integer size) and increments the address register of the output array by 4 to points to point to the first result to be inserted. A new accumulator register is created to hold temporary output results across iterations. The loop code increments the input array by 4 to point to its first element. An index register is set to 0 to signify the first iteration of the while loop. The loop begins by subtracting the input array size, n, from the index, which will eventually lead to a subtraction i - n = 0 when i = n, which stops the loop by jumping to the final instruction, labelled loop end. Otherwise, the current array element is loaded into a temporary register where a *mipsLoad* auxiliary function is used to determine how large the element is depending on whether it is a boolean or an integer. The array pointer is then incremented for the next iteration. Now, the current accumulator value is stored in the current output array index, pointed to by an address register. The output array is also incremented with an appropriate byte offset depending on the element type. Finally, the mutually recursive 'applyFunArg' function is used to apply the binary function on the current accumulator and current array input element pointed to by the temporary register, which becomes the new last computed result to be stored in the accumulator register. This is repeated by jumping back up to the beginning loop label until the condition fails, meaning we have iterated through all input array elements.

## Feature 3: Optimizations

This section describes the optimizations performed to Fasto expressions (i.e., abstract syntax trees) before compiling down to MIPS machine code. Namely, it describes *copy propagation*, *constant folding*, and *dead binding removal* optimizations. The aim of optimizing a compiler is to transform a program to a semantically equivalent output program that uses fewer resources (i.e., memory) and/or executes faster (i.e., cpu). Code optimization problems are usually NP-complete (i.e., no efficient polynomial time algorithm exists to find an optimal solution) or even undecidable (i.e., whether or not a solution exists is unknown). We generally strive to improve the program by simplifying it with heuristics and consider any improvement useful regardless of whether it is optimal or not. Thus, we now present a set of optimization passes that take Fasto programs as input and transform them into new programs that compute the same results, possibly more efficiently.

### Copy propagation

Copy propagation is the process of replacing unwanted declarations in e.g. variables, array indices, and let bindings. Notice, we detect propagatees (variables and constants to be propagated) at the level of let-binding expressions, and bind them in a symbol table. Thus, we assume unique variable names and do not handle shadowing.

#### `Var`

For variables, we perform copy propagation by looking up the variable name in the symbol table on line 4, and if it exists, we propagate the existing variable or constant on lines 5-6.

```
1 let rec copyConstPropFoldExp (vtable: VarTable) (e: TypedExp) =
2     match e with
3     | Var (name, pos) ->
4         match SymTab.lookup name vtable with
5             | Some (VarProp x) -> Var (x, pos)
6             | Some (ConstProp x) -> Constant(x, pos)
7             | None -> e
```

#### `Index`

For array indices, we perform the copy propagation optimization similar to variable names by looking up the array name in the symbol table on line 5. If it exists, we propagate the existing array index on line 6 but with the optimized index expression from line 4. Otherwise, we return the array index unchanged.

```
1 let rec copyConstPropFoldExp (vtable: VarTable) (e: TypedExp) =
2     match e with
3     | Index (name, e, t, pos) ->
4         let e' = copyConstPropFoldExp vtable e // optimize index
          ↪  expression e prehandedly
5         match SymTab.lookup name vtable with // Only copy-propagate
          ↪  variables for indexing
6             | Some (VarProp x) -> Index (x, e', t, pos)
7             | _ -> Index(name, e, t, pos)
```

### Let

For let expressions, *let* $x = a$ *in body*, we add a new binding *name* $\mapsto$ *var a* to the symbol table, optimize the body with the new symbol table, and build and return the optimized let expression. For example, *let* $y = x$ *in* $z = 3 + y$ is semantically equivalent to *let* $z = 3 + x$. On line 4, we recursively find the optimized declaration. If it is a variable, constant or let expression, we optimize it as follows. Firstly, if the declaration is a variable, *let* $x = a$, we add the variable to the symbol table and optimize the body of the let expression using that symbol table instead. Secondly, if it is a constant, e.g. *let* $x = 5$, we also add it as a propagatee to our symbol table, and optimize the body of the let expression with that. Finally, if the declaration in the let expression is itself a let expression, we swap the names and bodies, which yields a semantically equivalent expression.

```
1  let rec copyConstPropFoldExp (vtable: VarTable) (e: TypedExp) =
2      match e with
3      | Let (Dec (name, e, decpos), body, pos) ->
4          let e' = copyConstPropFoldExp vtable e
5          match e' with
6              | Var (varname, _) -> // let x = a
7                  let vtable' = SymTab.bind name (VarProp varname) vtable
8                  let body' = copyConstPropFoldExp vtable' body
9                  Let(Dec (name, e', decpos), body', pos)
10             | Constant (value, _) -> // let x = 5
11                 let vtable' = SymTab.bind name (ConstProp value) vtable
12                 let body' = copyConstPropFoldExp vtable' body
13                 Let (Dec (name, e', decpos), body', pos)
14             | Let (Dec (name2, e2, decpos2), body2, pos2) -> // let y = (let x = e1 in e2) in e3 restructured recursively to semantically-equivalent let x = e1 in let y = e2 in e3
15                 copyConstPropFoldExp vtable (Let (Dec (name2, e2, decpos), Let (Dec (name, body2, decpos2), body, pos2), pos))
16             | _ -> (* Fallthrough - for everything else, do nothing *)
17                 let body' = copyConstPropFoldExp vtable body
18                 Let (Dec (name, e', decpos), body', pos)
```

## Constant folding

Constant folding is the process of identifying and evaluating constant expressions at compile time rather than at runtime. For example, $i = 10 * 10$ can be simplified to $i = 100$. Further, constant propagation is the process of substituting the values of known constants into expressions at compile time.For example, if $x = 14$ and $y = x - 4$, then propagating x yields $y = 14 - 4 = 10$. Please consult *CopyConstPropFold.fs* for our code.

In our case, we can either propagate a variable name in a let binding or a constant value, which is modelled in the compiler domain with a "propagatee" type that we keep track of in a symbol table:

```
type Propagatee =
    ConstProp of Value
  | VarProp   of string
type VarTable = SymTab.SymTab<Propagatee>
```

The *copyConstPropFoldExp* function is modified to handle constant folding optimizations on typed Fasto expressions by transforming them into new programs that are used during MIPS machine code generation. We now take a look at the particular case of multiplication and logical conjunction.

`Multiplication`

Given an expression on the form $e = x * y$, we apply the following optimizations. Firstly, we make sure to optimize (i.e., reduce or simplify) the operands, $e1, e_2$ recursively on lines 4-5. Given the optimized operand expressions, we now apply the following optimization rules on lines 6-12 if both or a mix of the operands are constants where integers $x$ and $y$ are baked into $e1'$ and $e2'$ respectively.

Firstly, if $x \in \mathbb{Z}$ and $y \in \mathbf{Z}$, then we can simply replace the multiplication expression $x*y$ with a constant containing the result of the multiplication, as shown on line 7. Secondly, if either operand is 0, we know that $x*0 = 0*y = 0$ so we replace any multiplication expression with 0 as operand by a zero constant on lines 8-9. Thirdly, if we have a multiplication expressions on the form $x * 1 = 1 * x = x$, then we replace the multiplication by the optimized operand that is not equal to 1 on lines 10-11. Finally, if none of these scenarios are present we simply return the multiplication expression where both operands have been optimized recursively.

```
1  let copyConstPropFoldExp (vtable: VarTable) (e: TypedExp) =
2      match e with
3      | Times (e1, e2, pos) ->
4          let e1' = copyConstPropFoldExp vtable e1
5          let e2' = copyConstPropFoldExp vtable e2
6          match (e1', e2') with
7              | (Constant(IntVal x, _), Constant(IntVal y, _)) -> Constant
               ↪  (IntVal (x * y), pos)
8              | (Constant (IntVal 0, _), _) -> Constant (IntVal 0, pos)
9              | (_, Constant (IntVal 0, _)) -> Constant (IntVal 0, pos)
10             | (Constant (IntVal 1, _), _) -> e2'
11             | (_, Constant (IntVal 1, _)) -> e1'
12             | _ -> Times(e1', e2', pos)
```

`Logical Conjunction`

Given an expression on the form $xANDy$, we use the following conjunctive identity rules to simply the expression where T is true and F is false: $T\ AND\ x\ =\ x\ =\ x\ AND\ T$ and $F\ AND\ x\ =\ F\ =\ x\ AND\ F$. Again, we first optimize the operands on lines 4-5 before optimizing the logical conjunction between them. Firstly, if either operand is already true, we return the other operand on lines 7-8. Secondly, if either operand is false, we simply return false on lines 9-10. Finally, if none of these cases occur, we simply return the conjunction between the recursively optimized operands.

```
1  let copyConstPropFoldExp (vtable: VarTable) (e: TypedExp) =
2      match e with
3      | And (e1, e2, pos) ->
4          let e1' = copyConstPropFoldExp vtable e1
5          let e2' = copyConstPropFoldExp vtable e2
6          match (e1', e2') with
7              | (Constant(BoolVal true, _), _) -> e2'
8              | (_, Constant (BoolVal true, _)) -> e1'
9              | (Constant (BoolVal false, _), _) -> Constant (BoolVal
               ↪  false, pos)
10             | (_, Constant (BoolVal false, _)) -> Constant (BoolVal
               ↪  false, pos)
11             | _ -> And (e1', e2', pos)
```

### Dead-binding removal

Dead code elimination (i.e., DCE) is an optimization that removes code, which does not affect the output of the program. This mainly helps shrink the program size but may also reduce the running time of a program. Dead code usually includes unreadable code that can never be executed and variables (e.g. bindings) that are not ultimately used in the program. Again, we identify unused bindings in variables, array indices, and let bindings. This time however, we use a symbol table that only keeps track of used names and has no values, which is denoted with a type $typeDBRtab = SymTab.SymTab < unit >$ in *DeadBindingRemoval.fs.*

In this optimization, the input is a Fasto expression to be optimized by removing dead bindings. The function takes a typed Fasto program and the result is a three-tuple denoting whether the expression contains io, the symbol table that contains names used in the expressions, and the optimized typed expression.

### Variable

For variables, we assume that names cannot contain IO (i.e., write or read) so we return false, a fresh symbol table containing only the name of the variable, and the typed variable itself.

```
let rec removeDeadBindingsInExp (e : TypedExp) : (bool * DBRtab * TypedExp) =
    match e with
    | Var (name, pos) ->
        (false, SymTab.fromList[(name, ())], Var (name, pos))
```

### Array Index

For array indices, it is similar to variables, except we also need to recursively optimize the expression 'e' and propagate its results. Thus we propagate the check of whether the index expression contains io, a symbol table that combined any variable name uses in the index expression with the array name itself added to it, and the typed expression containing the optimized index expression.

```
let rec removeDeadBindingsInExp (e : TypedExp) : (bool * DBRtab * TypedExp) =
    match e with
    | Index (name, e, t, pos) ->
       let (io, uses, e') = removeDeadBindingsInExp e
       (io, SymTab.combine (SymTab.fromList[(name, ())]) uses, Index (name, e', t, pos))
```

### Let Expression

In the let expression, we recursively check any dead bindings in the body of the let-binding. If the name of the let declaration is not used in the body, we just return the optimized result of the body. Otherwise, we recursively process the binded expression 'e' in the let declaration, and join the two results (i.e., 'e' and 'body'). This means checking whether either one or both exhibit IO operations, joining their used names, and returning the let expression with an optimized declaration and body expression.

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

## Testing

During development, the test script has been run with various suitable flags. We have extended the provided suite of tests with additional tests for arithmetic, logical and array combinator operations. The arithmetic and boolean tests can be found in the `tests/arithmeic/` and `tests/boolean/` directories, respectively. The array combinator tests are in the root test folder with the rest of the tests. We are aware, that two of the boolean tests fail with a `DivideByZero` error when running them with optimizations enabled. This is due to short-circuiting not being handled in the `copyConstPropFold.fs` by neither our implementation of the `And` operation nor the provided implementation of the `Or` operation. Also, we do not handle any test case scenarios where shadowing of let bindings affect the final result.

## Conclusion

In this group project assignment, we have successfully managed to complete the handed out partial implementation to the Fasto compiler, written in F# as host language and MIPS as target language. We extended the Fasto compiler and interpreter to be able to handle integer multiplication and division, boolean literals, boolean binary operators `&&` and `||`, boolean negation, arithmetic negation, array combinators *scan*, *replicate* and *filter*. Further, we improved the optimizations of Fasto code by adding in copy propagation, constant folding and dead-binding removal. Ultimately, the project requires us to extend all compiler phase (files), including lexing, parsing, interpretation, type checking, machine code generation and optimizations. During the process, tests have been added to validate that arithmetic, booleans, array combinators, and optimizations work as intended. Notice, we have not implemented the optional array comprehension, nor have we addressed the issue of shadowing of let bindings, which could be solved by modifying the variable name symbol table on the fly to avoid name conflicts. Thus, we do not pass the tests, `comprehension.fo` and `negate.fo`. Our test for negate fails, because it prints out a `true`, where it should print out a `false`. Subsequent testing revealed that this was not a problem with how we handle negation, but rather due to shadowing of let bindings present after code optimisation, when inlining is performed, followed by copy propagation and constant folding. In `negate.fo`, the let bindings in the function `write_nl`, are optimised in such a way, that overshadowing occurs. We have not fixed this.