Fasto - Group Project

Adit (hjg708), Naomi (xng137), Nikolaj (bxz911)

IPS

June 7, 2023

Contents

1 Task 1

In this task, we extended the implementation of the FASTO compiler to include additional operators and boolean literals. The purpose of this task was to train us in the inner working of a compiler and that the program can interpret and translate correctly, while also rejecting non valid inputs.

1.1 Lexer.fsl

In the Lexer.fsl file we modified the lexer to recognize the new operators: `*, /, &&, ||, not,` ~ and boolean literals: `true, false`. These tokens were added to the token rule.
For boolean operations:

```
let keyword (s, pos) =
     match s with
       ...
       | "true"          -> Parser.TRUE pos
       | "false"         -> Parser.FALSE pos
       | "not"           -> Parser.NOT pos
```

For the other operations:

```
rule Token = parse
    ...
    | '*'                 { Parser.TIMES  (getPos lexbuf) }
    | '/'                 { Parser.DIVIDE (getPos lexbuf) }
    | '~'                 { Parser.NEGATE (getPos lexbuf) }
    | "&&"                { Parser.AND    (getPos lexbuf) }
    | "||"                { Parser.OR     (getPos lexbuf) }
    | ';'                 { Parser.SEMICOLON (getPos lexbuf) }
```

1.2 Parser.fsp

In the Parser.fsp file we updated it so that it is now able to handle the new expressions involving the added operators and literals. We extended the grammar by adding the new expressions. The new expressions are implemented as follows:

```
Exp :
    ...
    | TRUE             { Constant (BoolVal true, $1) }
    | FALSE            { Constant (BoolVal false, $1) }
    | Exp TIMES Exp    { Times($1, $3, $2) }
    | Exp DIVIDE EXP   { Divide($1, $3, $2) }
    | NOT Exp          { Not ($2, $1) }
    | NEGATE Exp       { Negate ($2, $1) }
    | Exp AND Exp      { And($1, $3, $2) }
    | Exp OR Exp       { Or($1, $3, $2) }
```

We decide if an operator is left-, right- or non-associative from the assignment text. We assign the operators precedence to determine their order of evaluation. Operators with higher precedence are evaluated first. When looking at precedence in the code, we read from the bottom and up. The operators are organized in the following precedence order:

- `NEGATE`: This operator has the highest precedence and is non-associative. It is used for numerical negation (~).
- `TIMES` and `DIVIDE`: These operators have the second highest precedence and are left-associative. They are used for multiplication (*) and division (/) operations.
- `PLUS` and `MINUS`: These operators have the third highest precedence and are left-associative. They are used for addition (+) and subtraction (-) operations.

- `DEQ` and `LTH`: These operators have the fourth highest precedence and are left-associative. They are used for equality (==) and less than (<) comparisons.
- `NOT`: This operator has the fifth highest precedence and is non-associative. It is used for logical negation (not).
- `AND`: This operator have the third lowest precedence and is left-associative. It is used for the logical conjunction (&&) operation.
- `OR`: This operator have the second lowest precedence and is left-associative. It is used for the logical disjunction (||) operations´.
- `IF` and `LET`: These operators have the lowest precedence and are non-associative. They are used for the if and let operations.

The precedence sequence can be seen in appendix 1.

We implemented `alt_Decs` as a nonterminal in Parser.fsp representing a sequence of declarations. It is used to handle multiple variable declarations within a `Let` expression.
The nonterminal `alt_Decs` is defined as follows:

```
    alt_Decs : alt_Dec SEMICOLON alt_Decs { $1 :: $3 }
             | alt_Dec { $1 :: [] }
    ;
```

This rule allows for one or more declarations separated by semicolons. We then incorporated `alt_Decs` into the `LET` function. We modified the production rule for the `Let` expression it looks as follows:

```
Exp :
    ...
    | LET alt_Decs IN Exp %prec letprec
      { List.foldBack (fun dec acc -> Let (dec, acc, $1)) $2 $4 }
```

Here, the `LET` keyword is followed by a sequence of declarations represented by `alt_Decs`. The `List.foldBack` funtion is used to construct a nested `Let` expression by iterating over the declarations in reverse order and accumulating them.

1.3 Interpreter.fs

In `Interpreter.fs` we updated the file so that it now evaluates the expressions correctly. We implemented the necessary evaluation logic for the new operators. The interpreter ensures that the boolean operators of `&&` and `||` are implemented correctly, meaning that the right-hand operand is only evaluated if the left-hand operand is true or false, respectively. Division by zero is caught and reported as an error accompanied by a relevant error message along with a source location of the error.

Implementation of `Divide` which is similar to `Times`, but in `Divide` we made sure to raise an error message in the case of a division by zero. The implementation for `Times` can be found in appendix 5.2.

```
    | Divide(e1, e2, pos) ->
        let res1   = evalExp(e1, vtab, ftab)
        let res2   = evalExp(e2, vtab, ftab)
        match (res1, res2) with
          | (IntVal _, IntVal 0) ->
            reportWrongType "right operand of / was 0"
            Int res2 (expPos e2)
          | (IntVal n1, IntVal n2) -> IntVal (n1/n2)
          | (IntVal _, _) -> reportWrongType "right operand of /"
            Int res2 (expPos e2)
          | (_, _) -> reportWrongType "left operand of /"
            Int res1 (expPos e1)
```

Implementation of And which is similar to Or. The implementation for Or can be found in appendix 5.3.

```
    | 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 false -> BoolVal false
              | BoolVal true -> BoolVal true
              | _ -> reportWrongType "right operand of &&"
                Bool res2 (expPos e2)
          | _ -> reportWrongType "left operand of &&"
            Bool res1 (expPos e1)
```

Implementation of Not:

```
    | Not(e1, pos) ->
        let res1   = evalExp(e1, vtab, ftab)
        match (res1) with
          | BoolVal false -> BoolVal true
          | BoolVal true -> BoolVal false
          | _ -> reportWrongType "operand of not" Bool res1
            (expPos e1)
```

Implementation of Negate:

```
    | Negate(e1, pos) ->
        let res1   = evalExp(e1, vtab, ftab)
        match (res1) with
          | IntVal n1 -> IntVal (0 - n1)
          | _ -> reportWrongType "can't negate value"
            (valueType res1) res1 pos
```

1.4 TypeChecker.fs

In the TypeChecker.fs file we extended the implementation so that it now can handle the new operators and literals. We added type checking rules for the expressions involving these additions. The type checker ensures that the operands of arithmetic operators have compatible types and boolean operators are applied to boolean expressions.

Implementation of Times which is similar to Divide. The implementation for Divide can be found in appendix 5.4.

```
    | Times (e1, e2, pos) ->
        let (e1_dec, e2_dec) = checkBinOp ftab vtab
        (pos, Int, e1, e2)
        (Int, Times (e1_dec, e2_dec, pos))
```

Implementation of And which is similar to Or. The implementation for Or can be found in appendix 5.5.

```
    | And (e1, e2, pos) ->
        let (e1_dec, e2_dec) = checkBinOp ftab vtab
        (pos, Bool, e1, e2)
        (Bool, And (e1_dec, e2_dec, pos))
```

Implementation of Not which is similar to Negate. The implementation for Negate can be found in appendix 5.6.

```
    | Not (e1, pos) ->
        let (t1, e1_dec) = checkExp ftab vtab e1
        match t1 with
          | Bool -> (Bool, Not (e1_dec, pos))
          | _ -> reportTypeWrong "argument of not " Bool t1 pos
```

1.5 CodeGen.fs

In the CodeGen.fs file we modified the RISC-V to give the appropriate instructions for the new operators. This modification ensures that the generated RISC-V code correctly performs arithmetic operations and handles boolean expressions.

1.5.1 Arithmetic operations

For arithmetic operations such as multiplication, and division, the code generator generates the corresponding RISC-V instructions. For example, the `*` operator is translated into the `MUL` instruction, and the `/` operator is translated into the `DIV` instruction. `Divide` and `times` was mostly copied from already implemented code, as we "pattern matched" from `minus` and `plus`.

```
  | 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 @ [MUL (place,t1,t2)]
```

We changed 'Divide' so that it can now handles division by zero

```
  | Divide (e1, e2, pos) ->
      let t1 = newReg "divide_L"
      let t2 = newReg "divide_R"
      let zero = newReg "zero"
      let divLabel = newLab "divLabel"
      let code1 = compileExp e1 vtable t1
      let code2 = compileExp e2 vtable t2
      code1 @ code2 @
      [ LI (zero, 0)
      ; BNE (t2, zero, divLabel )]
      @ [LA (Ra1, "m.DivZero")]
      @ [JAL ("p.RuntimeError", [Ra1])]
      @ [LABEL divLabel]
      @ [DIV (place,t1,t2)]
```

1.5.2 Boolean operators

When it comes to boolean operators, the code generator generates conditional branch instructions based on the expressions. For the `&&` operator, the code generator gives a sequence of instructions that perform relevant evaluations. If the left expression is false, the code generator generates a branch instruction to skip the evaluation of the right expression. Similarly, for the `||` operator, the code generator generates a branch instruction to skip the right expression if the left expression is true.

```
  | And (e1, e2, pos) ->
      let t1 = newReg "and_L"
      let t2 = newReg "and_R"
      let t3 = newReg "falseReg"
      let code1 = compileExp e1 vtable t1
      let code2 = compileExp e2 vtable t2
```

```
      let code3 = compileExp (Constant (IntVal 0, pos)) vtable t3
      let falseLabel = newLab "false"
      code1 @ code3 @
      [ LI (place, 0);
        BEQ (t1, t3, falseLabel) ]
      @ code2
      @ [ BEQ(t2, t3, falseLabel);
        LI (place, 1);
        LABEL falseLabel]
```

The complete implementation for `Or` is in the appendix 5.7.

```
   | Or (e1, e2, pos) ->
      ...
      code1 @ code3 @
      [ LI (place, 1);
        BEQ(t1, t3, trueLabel) ]
      @ code2
      @ [ BEQ(t2, t3, trueLabel);
        LI (place, 0);
        LABEL trueLabel ]
```

1.5.3 Negation operators

For logical not we implemented the operation using and numerical `XORI` instruction. For numerical not we used the `SUB` instruction, as you can turn a number into its negative (or positive) by subtracting it from 0.

```
  | Not (e1, pos) ->
      let t1 = newReg "not"
      let code1 = compileExp e1 vtable t1
      code1 @ [XORI (place, t1, int 1)]
```

```
  | Negate (e1, pos) ->
      let t1 = newReg "negate"
      let code1 = compileExp e1 vtable t1
      let zeroReg = newReg "zero"
      code1 @ [SUB (place, zeroReg, t1)]
```

1.6 Task 1 tests

We added more tests, to help further test our implementations, those are:

- and
- div
- divide_by_zero
- letin
- not
- or
- times

The code can be found in appendix 5.10.

1.6.1 Compiler

When we finished implementing the TODO's for Task 1 for the milestone submission, all the tests were successful except for comprehension.fo, filter.fo, filter-on-2darr.fo, multilet.fo, replicate.fo, scan.fo and short_circuit.fo.
We made `multilet.fo` successful by implementing `alt_Decs` as explained in 1.2 Parser. We made `short_circuit.fo` successful by changing the code in `CodeGen.fs` so that the first expression was evaluated first before we looked at the second expression. We implemented this principle in `And` by adding a third code (code3) that was false and ''ing code1 and code3 before looking if code1 was false (should go to falseLabel) or true (see code snippet in 1.5.2 Boolean operaters). `Or` was implemented similarly, except that code3 was true, and we used a trueLabel.

1.6.2 Interpreter

For the milestone submission, the interpreter test `short_circuit.fo` failed with stack overflow. We overcame this by changing the code in `Interpreter.fs` for `And` and `Or` by not using `evalExp` on `e2` before `e1` had been evaluated. Our implementations can be seen in 1.3 Interpreter.
All other tests except for `divide_by_zero` are successful. Testing `divide_by_zero` with the interpreter is unsuccessful, because we have different fail messages for the compiler and interpreter, but they both fail gracefully.

2 Task 2

2.1 Lexer.fsl

The lexer code has been updated to include the following keyword tokens:

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

These additions allow the lexer to recognize the keywords `replicate`, `filter`, and `scan` in the input code. Each keyword is paired with a corresponding token in the `Parser` module, which are `Parser.REPLICATE`, `Parser.FILTER`, and `Parser.SCAN`.

2.2 Parser.fsp

In Parser.fsp we implemented the tokens for `replicate`, `filter`, and `scan`:

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

These tokens correspond to the keywords `replicate`, `filter`, and `scan`. These tokens makes it so the parser can recognize and associate the keywords with the appropriate grammar rules during the parsing process. These grammar rules are as follows:

```
    Exp : ...
        | 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) }
```

These grammar rules define the syntax for the `replicate`, `filter`, and `scan` expressions.

2.3 Interpreter.fs

2.3.1 Replicate

For the interpreter module we implemented the `Replicate` case by evaluating the count and value expressions and using the `List.replicate` function to create a new list with the specified size of the given value. We then construct an `ArrayVal` with the replicated list and the type of the value. We also handled the case where the count evaluates to an integer value less than zero by raising an error.

```
| Replicate (nExp, eltExp, _, pos) ->
      let n    = evalExp(nExp, vtab, ftab)
      let elt  = evalExp(eltExp, vtab, ftab)
      match n with
        | IntVal size ->
            if size < 0 then raise
            (MyError("Negative size is now allowed " +
            ppVal 0 n, pos))
            else ArrayVal (List.replicate size elt, valueType elt)
        | otherwise -> raise
          (MyError("Expected number as argument: "
          + ppVal 0 n, pos))
```

2.3.2 Filter

In the `Filter` case, we used `evalExp` to evaluate the array expressions. If the predicate funtion of farg_ret_type is not a boolean we raise an error, if it is then we then use the `List.filter` function to filter the array elements based on the predicate function. We then create a new `ArrayVal`. If the predicate function return a non-boolean value, we raise an exception, this ensures that the input is an array.

```
| Filter (farg, arrexp, _, pos) ->
    let arr  = evalExp(arrexp, vtab, ftab)
    let farg_ret_type = rtpFunArg farg ftab pos
    if farg_ret_type <> Bool then
        raise (MyError("Predicate function should return a boolean"
        , pos))
    match arr with
        | ArrayVal (lst, tp1) ->
            let filtered_lst =
                List.filter (fun x -> match evalFunArg
                (farg, vtab, ftab, pos, [x]) with
                    | BoolVal b -> b
                    | _ -> reportWrongType "return value of
                      predicate function" Bool x pos
                ) lst
            ArrayVal (filtered_lst, tp1)
        | otherwise -> reportNonArray
          "2nd argument of \"filter\"" arr pos
```

2.3.3 Scan

We implemented the `Scan` operation by evaluating the initial value expression `nexp` and the array expression `arrexp`. We then check the return type of the function argument `farg`. The code applies the function argument to each pair of elements in the array, using the initial value as the starting point,and produces a new array using `List.scan` function. The resulting array is of the same type and length as the input array. If the input is not an array, an error exception is raised.

```
| Scan (farg, nexp, arrexp, tp, pos) ->
    let farg_ret_type = rtpFunArg farg ftab pos
```

```
      let init  = evalExp(nexp, vtab, ftab)
      let arr  = evalExp(arrexp, vtab, ftab)
      match arr with
        | ArrayVal (lst, tp1) ->
            ArrayVal (List.tail (List.scan (fun x y -> evalFunArg
            (farg, vtab, ftab, pos, [x; y])) init lst),
            farg_ret_type)
        | otherwise -> raise (MyError("Argument is not an array: "
          + ppVal 0 arr, pos))
```

2.4 TypeChecker.fs

2.4.1 Replicate

In `TypeChecker.fs` we implemented the `Replicate` expression by examining its sub-expressions. If the first sub-expression `n_type` is an integer, it will return an array of the type of the second sub-expression '`a_type`'. The decorated version of the expression `n_dec` is constructed using the decorated sub-expression `a_dec` and the inferred type `a_type`. If the first sub-expression is not an integer, we raise an error exception.

```
    | Replicate (n_exp, a_exp, _, pos) ->
        let (n_type  , n_dec  ) = checkExp ftab vtab n_exp
        let (a_type  , a_dec  ) = checkExp ftab vtab a_exp
        match n_type with
          | Int -> (Array a_type, Replicate
            (n_dec, a_dec, a_type, pos))
          | otherwise -> failwith "Type error"
```

2.4.2 Filter

For the `Filter` expression. We in line 2 checked the type of the array expression and extracted the element type from it. Then in line 9, we used the `checkFunArg` function to obtain the signature of the function argument `f`. We verified that the function argument has the correct type by comparing it with the element type of the array and checking if it returns a boolean value. If the types were compatible, we returned the decorated expression with the array element type (line 12). However in line 13, if any type incompatibility was detected during the checks, we raised an exception error. In line 20 the expression then return an array of the function argument type `f_arg_type`, along with the decorated version of the array expression `arr_dec`, and the inferred type `f_arg_type`.

```
1      | Filter (f, arr_exp, _, pos) ->
2          let (arr_type, arr_dec) = checkExp ftab vtab arr_exp
3          let elem_type =
4              match arr_type with
5                | Array t -> t
6                | other -> raise (MyError ("Filter: Argument not an
7                array", pos))
8          let (f', f_arg_type, f_res_type) =
9              match checkFunArg ftab vtab pos f with
10               | (f', res, [arg]) ->
11                   if arg = elem_type && res = Bool
12                   then (f', arg, res)
13                   else raise (MyError( "Filter: incompatible function
14                   type of " + (ppFunArg 0 f) + ": " + showFunType
15                   ([arg], res), pos))
16               | (_, res, args) ->
17                   raise (MyError ( "Filter: incompatible function
18                   type of " + ppFunArg 0 f + ": " + showFunType
19                   (args, res), pos))
20         (Array f_arg_type, Filter (f', arr_dec, f_arg_type, pos))
```

2.4.3 Scan

In our implementation of `Scan`, we extract the element type from the array type and check the compatibility of the function argument. If the function argument has a compatible type, we return an array of the element type, otherwise, we raise an exception error. The result is the decorated expression along with the inferred element type.

```
    | 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
             (Array elem_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))
```

2.5 CodeGen.fs

2.5.1 Replicate

The implementation allows us to dynamically allocate memory for the result array, replicates the value n times using a loop, and stores the replicated values in the allocated memory.

```
  | Replicate (n_exp, arr_exp, tp, (line, _)) ->
        let size_reg = newReg "size"
        let n_code = compileExp n_exp vtable size_reg
        let arr_reg = newReg "arr"
        let arr_code = compileExp arr_exp vtable arr_reg

        let addr_reg = newReg "addrg"
        let safe_lab = newLab "safe"
        let checksize = [ BGE (size_reg, Rzero, safe_lab)
                        ; LI (Ra0, line)
                        ; LA (Ra1, "m.BadSize")
                        ; J "p.RuntimeError"
                        ; LABEL (safe_lab)
                        ]
```

```
        let i_reg = newReg "i"
        let init_regs = [ ADDI (addr_reg, place, 4)
                        ; MV (i_reg, Rzero) ]

        let elem_size = getElemSize tp
        let loop_beg = newName "loop_beg"
        let loop_end = newName "loop_end"
        let loop_header = [ LABEL (loop_beg)
                          ; BGE (i_reg, size_reg, loop_end)
                          ]
        let loop_replicate = [ Store elem_size (arr_reg, addr_reg, 0)
                             ; ADDI (addr_reg, addr_reg, 4)
                             ]
        let loop_footer = [ ADDI (i_reg, i_reg, 1)
                          ; J loop_beg
                          ; LABEL loop_end
                          ]
        n_code
         @ arr_code
         @ checksize
         @ dynalloc (size_reg, place, tp)
         @ init_regs
         @ loop_header
         @ loop_replicate
         @ loop_footer
```

2.5.2 Filter

For the implementation we first declare our registers, but we removed them from the snippet to save space, the registers are all the variables ending in _reg, the entire filter can be found in appendix 5.8. The implementation allows us to filter the elements of the input array based on the function argument `f`, and retaining only those elements for which `f` evaluates true.

```
  | Filter (f, arr_exp, tp, pos) ->

      ...

      let alloc_code = dynalloc (size_reg, place, tp)
      let arr_code = compileExp arr_exp vtable arr_reg
      let loop_beg = newLab "loop_beg"
      let loop_end = newLab "loop_end"
      let filter_false = newLab "filter_false"

      let get_size = [ LW (size_reg, arr_reg, 0) ]
      let init_regs = [ ADDI (addr_reg, place, 4)
                      ; MV (i_reg, Rzero)
                      ; ADDI (tmp_reg, arr_reg, 4)
                      ; ADDI (j_reg, Rzero, 0)
                      ]

      let tp_size = getElemSize tp
      let loop =
             [ LABEL (loop_beg)
             ; BGE (i_reg, size_reg, loop_end)
             ; Load tp_size (bool_reg, tmp_reg, 0)
             ; Load tp_size (res_addr_reg, tmp_reg, 0)
             ; ADDI (tmp_reg, tmp_reg, elemSizeToInt tp_size)
             ]
```

```
            @ applyFunArg(f, [bool_reg], vtable, res_reg, pos)
            @ [ BEQ (res_reg, Rzero, filter_false)
            ; Store tp_size (res_addr_reg, addr_reg, 0)
            ; ADDI (addr_reg, addr_reg, elemSizeToInt tp_size)
            ; ADDI (j_reg, j_reg, 1)
            ; LABEL filter_false
            ; ADDI (i_reg, i_reg, 1)
            ; J loop_beg
            ; LABEL loop_end
            ; SW (j_reg, place, 0)
            ]

      arr_code
       @ get_size
       @ alloc_code
       @ init_regs
       @ loop
```

2.5.3 Scan

For the implementation we first declare our registers and labels, as with Filter we removed them to save space, the entire scan can be found in appendix 5.9. The implementation allows us to generate the necessary instructions to perform a scan operation by maintaining the state of registers and iterating through the input array through a loop.

```
  | Scan (binop, acc_exp, arr_exp, tp, pos) ->

      ...

      let arr_code = compileExp arr_exp vtable arr_reg
      let header1 = [ LW(size_reg, arr_reg, 0) ]

      let init_regs = [ ADDI (addr_reg, place, 4) ]

      let acc_code = compileExp acc_exp vtable acc_reg

      let loop_code =
              [ ADDI(arr_reg, arr_reg, 4)
              ; MV(i_reg, Rzero)
              ; LABEL(loop_beg)
              ; SLT(tmp_reg, i_reg, size_reg)
              ; BEQ(tmp_reg, Rzero, loop_end)
              ]
      let elem_size = getElemSize tp
      let load_code =
              [ Load elem_size (tmp_reg, arr_reg, 0)
              ; ADDI (arr_reg, arr_reg, 4)
              ]
      let store_code =
              [ Store elem_size (acc_reg, addr_reg, 0)
              ; ADDI (addr_reg, addr_reg, 4)
              ]
      let apply_code =
              applyFunArg(binop, [acc_reg; tmp_reg],
              vtable, acc_reg, pos)
      let loop =
              [ ADDI(i_reg, i_reg, 1)
              ; J loop_beg
```

```
                ; LABEL loop_end
                ]

       arr_code
         @ header1
         @ dynalloc (size_reg, place, tp)
         @ init_regs
         @ acc_code
         @ loop_code
         @ load_code
         @ apply_code
         @ store_code
         @ loop
```

2.6 Task 2 tests

2.6.1 Compiler and Interpreter

We ran the already created tests for `filter`, `scan` and `replicate`, and they were all successful when testing for the compiler and the interpreter.

So in addition to the test we performed in task 1, we have now examined all 30 test cases for both the compiler and interpreter mode, which in total is 60 tests. Among these tests three of them return as failed, two of them is for `Comprehension.fo` which should be tackled in task 4, and the other on is for `divide_by_zero.fo` which failed gracefully.

3 Task 3

3.1 CopyConstPropFold.fs

For the implementation of task 3 we look at `CopyConstPropFold.fs`, first we had to finish implementing `Var`, `Index` and `Let`. For `Var (name, pos)` we used match case using `SymTab.lookup` in line 3 to check if there is an entry for the variable `name`. In line 4 if an entry is a propagated variable `VarProp x`, replace the expression with `Var (x, pos)`, where x is the variable name. In line 5 if an entry is a propagated constant `ConstProp y`, replace the expression with Constant (y, pos), where y is the constant value. In line 6 if no entry is found or the entry is not a propagated value, the expression remains unchanged.

We do the same thing in `Index` but we first optimized `e` using `copyConstPropFoldExp` in line 8. But unlike `Var` we only evaluate propagated variable and not for propagated constant.

For let-bindings line 16 we again use `copyConstPropFoldExp` to optimized `e`. We then checks the optimized `e` using match cases in line 17. For the `Var` case we associated the variable name with a propagated variable in the variable table using SymTab.bind. Lastly on line 22 we then create a new let-binding with an optimized expression and body. This process is the same for the `Const` case, just that we instead use constant instead of variable.

If the optimized `e` is neither a variable nor a constant, it means that no optimizations were possible. We then create a new let-binding using the optimized expression **e2'** and body **body2'** without modifying the variable table.

```
1      match e with
2          | Var (name, pos) ->
3              match SymTab.lookup name vtable with
4                  | Some (ConstProp x) -> Constant(x, pos)
5                  | Some (VarProp y)   -> Var(y, pos)
6                  | _                  -> Var(name, pos)
```

```

        | Index (name, e, t, pos) ->
          let e' = copyConstPropFoldExp vtable e
          match SymTab.lookup name vtable with
              | Some (VarProp x)   -> Index(x, e', t, pos)
              | Some (ConstProp y) -> failwith "Impossible"
              | _                  -> Index(name, e', t, pos)

        | Let (Dec (name, e, decpos), body, pos) ->
          let e' = copyConstPropFoldExp vtable e
          match e' with
              | Var (varname, _) ->
                let vtable' = SymTab.bind name (
                    VarProp varname) vtable
                let body' = copyConstPropFoldExp vtable' body
                Let (Dec (name, e', decpos), body', pos)

              | ...

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

              | ...
```

For times-binding we again use `copyConstPropFoldExp` to optimized the expressions. The implementation follows a pattern of checking for constant values and performing simplifications based on the algebraic properties of multiplication. This helps optimizing the expression and potentially eliminating unnecessary computations.

The same principle is used to implement `And`. The implementation optimize and simplify the and-binding by using the boolean properties of and operation. Which is that if one of the expressions are false then the operation will also return false.

```
        | Times (e1, e2, pos) ->s
          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 0, _), _) ->
                Constant (IntVal 0, pos)
              | (_, Constant (IntVal 0, _)) ->
                Constant (IntVal 0, pos)
              | (_, Constant (IntVal 1, _)) -> e1'
              | (Constant (IntVal 1, _), _) -> e2'
              | (_, Constant (IntVal -1, _)) -> Negate (e1', pos)
              | (Constant (IntVal -1, _), _) -> Negate (e2', pos)
              | _ -> Times (e1', e2', pos)

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

```
                  | (Constant (BoolVal false, _), _) ->
                      Constant (BoolVal false, pos)
                  | (_, Constant (BoolVal false, _)) ->
                      Constant (BoolVal false, pos)
                  | _ -> And (e1', e2', pos)
```

3.2 DeadBindingRemoval.fs

For our implementation of DeadBindingRemoval we first implement the dead binding removal case for `Var`. By using `recordUse` function we can record the use of the variable `name` in the newly created symbol table `stab`. This then add a new entry in the symbol table if the variable is not already present. It returns a tuple containing a flag (false), the updated symbol table `stab`, and the optimized expression `Var (name, pos)`.

For the `Index` case we process the index expression recursively and updating the symbol table. We record the use of `name` in the expression `e`. By using the record we can optimize the process to accurately determine whether the variable `name` is used and we then can remove dead bindings accordingly. It returns a tuple containing the IO flag `eio`, the temporary symbol table `tempStab`, and the optimized expression `Index(name, e', t, pos)`.

For the `Let` case we recursively process the expressions e and body using the `removeDeadBindingsInExp` function and store the results in `(eio, euses, e')` and `(bodyio, bodyuses, body')`, respectively. We then check if the variable `name` is used in `bodyuses` or if `eio` is true. If either condition is true, it creates a new Let-expression with the optimized subexpressions and combines the symbol tables `euses` and `bodyuses`, while removing the variable name from `bodyuses`. Otherwise it returns the optimized body expression and the corresponding IO flag and symbol table. It returns the same as Var and Index.

```
let rec removeDeadBindingsInExp
(e : TypedExp) : (bool * DBRtab * TypedExp) =
    match e with
        ...
        | Var (name, pos) ->
            let stab = recordUse name (SymTab.empty())
            (false, stab, Var (name, pos))
        ...
        | Index (name, e, t, pos) ->
            let (eio, euses, e') = removeDeadBindingsInExp e
            let tempStab = recordUse name euses
            (eio, tempStab, Index(name, e', t, pos))
        | 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
                (bodyio || eio,
                 SymTab.combine euses (SymTab.remove name bodyuses) ,
                 Let (Dec (name, e', decpos), body', pos))
            else
                (bodyio, bodyuses, body')
```

3.3 Task 3 tests

3.3.1 Optimization

For the optimization test, we applied a total of 30 tests. Out of these 30 tests, two of them returned as failed. One of them is `Comprehension.fo` which should be tackled in task 4 and the other is `Inline_shadom.fo`. The Inline_shadow test is incorrect but it is optional so therefore we decided not to solve it.

4 Task 4

After careful consideration and discussion, we, as a group, have decided not to complete the optional task 4 due to time constraints. As a result, the `comprehension.fo` test will fail for the compiler, interpreter, and optimization tests.

5 Appendix

5.1 Precedence

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

5.2 Interpreter.fs - Times

```
    | 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)
          | (IntVal _, _) -> reportWrongType
            "right operand of *" Int res2 (expPos e2)
          | (_, _) -> reportWrongType
            "left operand of *" Int res1 (expPos e1)
```

5.3 Interpreter.fs - Or

```
    | 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 false -> BoolVal false
            | BoolVal true -> BoolVal true
            | _ -> reportWrongType "right operand of ||"
                Bool res2 (expPos e2)
        | _ -> reportWrongType "left operand of ||"
            Bool res1 (expPos e1)
```

5.4 TypeChecker.fs - Divide

```
    | Divide (e1, e2, pos) ->
        let (e1_dec, e2_dec) = checkBinOp ftab vtab
            (pos, Int, e1, e2)
            (Int, Divide (e1_dec, e2_dec, pos))
```

5.5 TypeChecker.fs - Or

```
    | Or (e1, e2, pos) ->
        let (e1_dec, e2_dec) = checkBinOp ftab vtab
        (pos, Bool, e1, e2)
        (Bool, Or (e1_dec, e2_dec, pos))
```

5.6 TypeChecker.fs - Negate

```
    | Negate (e1, pos) ->
        let  (t1, e1_dec) = checkExp ftab vtab e1
        match t1 with
          | Int -> (Int, Negate (e1_dec, pos))
          | _ -> reportTypeWrong "argument of negate " Int t1 pos
```

5.7 CodeGen.fs - Or

```
   | Or (e1, e2, pos) ->
       let t1 = newReg "or_L"
       let t2 = newReg "or_R"
       let t3 = newReg "trueReg"
       let code1 = compileExp e1 vtable t1
       let code2 = compileExp e2 vtable t2
       let code3 = compileExp
         (Constant (IntVal 1, pos)) vtable t3
       let trueLabel = newLab "true"
       code1 @ code3 @
       [ LI (place, 1);
         BEQ(t1, t3, trueLabel) ]
       @ code2
       @ [ BEQ(t2, t3, trueLabel);
         LI (place, 0);
         LABEL trueLabel ]
```

5.8 CodeGen.fs - Filter

```
  | Filter (f, arr_exp, tp, pos) ->
       let arr_reg  = newReg "arr"
       let tmp_reg  = newReg "tmp"
       let bool_reg  = newReg "bool"
       let res_reg  = newReg "res"
       let res_addr_reg  = newReg "res_arg"
       let addr_reg  = newReg "addrg"
       let i_reg     = newReg "i"
       let j_reg     = newReg "j"
       let size_reg = newReg "size"
```

```
      let alloc_code = dynalloc (size_reg, place, tp)
      let arr_code = compileExp arr_exp vtable arr_reg
      let loop_beg = newLab "loop_beg"
      let loop_end = newLab "loop_end"
      let filter_false = newLab "filter_false"

      let get_size = [ LW (size_reg, arr_reg, 0) ]
      let init_regs = [ ADDI (addr_reg, place, 4)
                      ; MV (i_reg, Rzero)
                      ; ADDI (tmp_reg, arr_reg, 4)
                      ; ADDI (j_reg, Rzero, 0)
                      ]

      let tp_size = getElemSize tp
      let loop =
              [ LABEL (loop_beg)
              ; BGE (i_reg, size_reg, loop_end)
              ; Load tp_size (bool_reg, tmp_reg, 0)
              ; Load tp_size (res_addr_reg, tmp_reg, 0)
              ; ADDI (tmp_reg, tmp_reg, elemSizeToInt tp_size)
              ]
              @ applyFunArg(f, [bool_reg], vtable, res_reg, pos)
              @ [ BEQ (res_reg, Rzero, filter_false)
              ; Store tp_size (res_addr_reg, addr_reg, 0)
              ; ADDI (addr_reg, addr_reg, elemSizeToInt tp_size)
              ; ADDI (j_reg, j_reg, 1)
              ; LABEL filter_false
              ; ADDI (i_reg, i_reg, 1)
              ; J loop_beg
              ; LABEL loop_end
              ; SW (j_reg, place, 0)
              ]

      arr_code
       @ get_size
       @ alloc_code
       @ init_regs
       @ loop
```

5.9 CodeGen.fs - Scan

```
  | Scan (binop, acc_exp, arr_exp, tp, pos) ->
      let arr_reg  = newReg "arr"   (* address of array *)
      let size_reg = newReg "size"  (* size of input array *)
      let i_reg    = newReg "i"     (* loop counter *)
      let tmp_reg  = newReg "tmp"   (* several purposes *)
      let loop_beg = newLab "loop_beg"
      let loop_end = newLab "loop_end"
      let addr_reg = newReg "addrg"
      let acc_reg  = newReg "acc"

      let arr_code = compileExp arr_exp vtable arr_reg
      let header1 = [ LW(size_reg, arr_reg, 0) ]

      let init_regs = [ ADDI (addr_reg, place, 4) ]

      let acc_code = compileExp acc_exp vtable acc_reg
```

```

      let loop_code =
              [ ADDI(arr_reg, arr_reg, 4)
              ; MV(i_reg, Rzero)
              ; LABEL(loop_beg)
              ; SLT(tmp_reg, i_reg, size_reg)
              ; BEQ(tmp_reg, Rzero, loop_end)
              ]
      let elem_size = getElemSize tp
      let load_code =
              [ Load elem_size (tmp_reg, arr_reg, 0)
              ; ADDI (arr_reg, arr_reg, 4)
              ]
      let store_code =
              [ Store elem_size (acc_reg, addr_reg, 0)
              ; ADDI (addr_reg, addr_reg, 4)
              ]
      let apply_code =
              applyFunArg(binop, [acc_reg; tmp_reg],
              vtable, acc_reg, pos)
      let loop =
              [ ADDI(i_reg, i_reg, 1)
              ; J loop_beg
              ; LABEL loop_end
              ]

      arr_code
        @ header1
        @ dynalloc (size_reg, place, tp)
        @ init_regs
        @ acc_code
        @ loop_code
        @ load_code
        @ apply_code
        @ store_code
        @ loop
```

5.10 New tests

and.fo:

```
fun bool and_func(bool n) =
  let res = write(n) in
  let tmp = write("\n") in
  res

fun bool main() =
  let n1 = and_func(true && true) in
  let n2 = and_func(false && false) in
  let n3 = and_func(false && true) in
  let n4 = and_func(true && false) in
  and_func(n1 && n2 && n3 && n4)
```

div.fo:

```
fun int div(int n) =
  10/n

fun int main() =
```

```
  let n = read(int) in
  write(div(n))
```

divide_by_zero.fo:

```
fun int divide(int n) =
  2/n

fun int main() =
  let n = read(int) in
  write(divide(n))
```

letin.fo:

```
fun bool main() =
  let x = not true in
  let y = x && true in
  let z = x && y in
  write(z)
```

not.fo:

```
fun bool not_func(bool n) =
  let res = write(n) in
  let tmp = write("\n") in
  res

fun bool main() =
  let n1 = not_func(not true) in   // false
  let n2 = not_func(not false) in  // true
  true
```

or.fo:

```
fun bool or_func(bool n) =
  let res = write(n) in
  let tmp = write("\n") in
  res

fun bool main() =
  let n1 = or_func(true || true) in
  let n2 = or_func(false || false) in
  let n3 = or_func(false || true) in
  let n4 = or_func(true || false) in
  or_func(n1 || n2 || n3 || n4)
```

times.fo:

```
fun int times(int n) =
  2*n

fun int main() =
  let n = read(int) in
  write(times(n))
```