Group Assignment

Compilers

Oliver Mikkel Winther Cristensen (klw741)

Mathias Brøndum Leroul (swn848)

Sebastian O. Jensen (gjx653)

Turn-in date: 20-12-2015

Contents

1 Introduction

We have been given the grand task of creating a small compiler, using the high level languages Fasto and SML as well as the low level machine language MIPS. Throughout this paper, we will document every code addition we make and if nothing is stated about the success of the code, it means that the code returns the expected output.

2 Implementation

Each implementation step is presented in the same order as the compiler structure: Lexer → Parser → Type Checker → Interpreter ∨ MIPS Code Generator

2.1 Task 1

Task 1 is the implementation of `true/false`, multiplication/division, `AND/OR` and finally `not/negate`.

2.1.1 `true` and `false`

To begin with, we're given the sub-task to implement the boolean literals `true` and `false`. We start out by creating the following test files:

File name	Code

`bool.fo`	`fun int boolTest(int a) = if a == 1`
	`then 1 else 0`
	
	`fun int main() =`
	`    let a = read(int) in`
	`    write(boolTest(a))`

`bool.in`	`1`
`bool.out`	`1`

Starting at the top of the compiler structure, the lexer is upgraded with the proper `true` and `false` keywords

`Lexer.lex`

`| "true"  => Parser.TRUE  pos`
`| "false" => Parser.FALSE pos`

Then we implement the `true/false` token type in our parser as well as add `true` and `false` to the expression list.

File name	Code
`Parser.grm`	`%token <(int*int)> TRUE FALSE`

`Parser.grm`	`| TRUE          { Constant (BoolVal true,  $1)}`
	`| FALSE         { Constant (BoolVal false, $1)}`

And finally we add the boolean constants to our MIPS code generator.

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

2.1.2 Multiplication and Division

Moving on to the implementation of multiplication and division, we again start out with creating test files for each of the two operators.

File name	Code

`Mult.fo`	`fun int Mult(int a) = a * 2`
	
	`fun int main() =`
	`    let a = read(int) in`
	`    write(Mult(a))`

`Mult.in`	`4`
`Mult.out`	`8`

`Div.fo`	`fun int Div(int a) = a / 2`
	
	`fun int main() =`
	`    let a = read(int) in`
	`    write(Div(a))`

`Div.in`	`128`
`Div.out`	`64`

After creating the test files, we swiftly move on to adding multiplication and division to the lexer.

```
Lexer.lex
```
```
| ‘*‘  { Parser.TIMES  (getPos lexbuf) }
| ‘/‘  { Parser.DIVIDE (getPos lexbuf) }
```

In the parser, we add `TIMES` and `DIVIDE` to the already created ”`PLUS MINUS DEQ EQ LTH`” token type. They’re given a proper precedence order, as well as added to the expression list.

File name	Code
`Parser.grm`	`%token <(int*int)> PLUS MINUS DEQ EQ LTH DIVIDE TIMES`

`Parser.grm`	`%left DIVIDE`
	`%left TIMES`

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

Moving on to the type checker, we make sure that we can properly check the used types, when doing multiplication and division, adding the cases to the `checkExp` function.

```
TypeChecker.sml
```

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

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

We go ahead and implement them in our interpreter, extending `evalExp` function with the proper cases.

```
Interpreter.sml
```

```
| evalExp ( Times(e1, e2, pos), vtab, ftab ) =
      let val res1   = evalExp(e1, vtab, ftab)
          val res2   = evalExp(e2, vtab, ftab)
      in  case (res1, res2) of
              (IntVal n1, IntVal n2) => IntVal (n1*n2)
            | _ => invalidOperands
                    "Multiplication on non-integral args: "
                    [(Int, Int)] res1 res2 pos
      end

| evalExp ( Divide(e1, e2, pos), vtab, ftab ) =
      let val res1   = evalExp(e1, vtab, ftab)
          val res2   = evalExp(e2, vtab, ftab)
      in  case (res1, res2) of
              (IntVal n1, IntVal n2) => IntVal (n1 div n2)
            | _ => invalidOperands
                    "Division on non-integral args: "
                    [(Int, Int)] res1 res2 pos
      end
```

Finally, cases for multiplication and division are added to the `compileExp` function in the MIPS code generator.

```
CodeGen.sml
```

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

2.1.3 AND and OR

For the AND and OR implementations, we add them as short-circuit, which means, that if the left hand side of the expression is evaluated to false, we immediately raise an error without evaluating the right hand side.
We again start out with creating test files for each of them.

File name | Code

and.fo
```
fun bool FAnd(bool a, bool b) = a && b

fun bool main() =
    let a = read(bool) in
    let b = read(bool) in
    let c = write(FAnd(a,a)) in
    let d = write(FAnd(b,b)) in
    let e = write(FAnd(a,b)) in
    write(FAnd(b,a))
```

and.in
```
1
0
```

and.out
```
truefalsefalsefalse
```

or.fo
```
fun bool orTest(bool a, bool b) = a || b

fun bool main() =
    let a = read(bool) in
    let b = read(bool) in
    let c = write(orTest(b,b)) in
    let d = write(orTest(a,a)) in
    let e = write(orTest(a,b)) in
    write(orTest(b,a))
```

or.in
```
1
0
```

or.out
```
falsetruetruetrue
```

With the test files in place, we can move on to the lexer.

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

In the parser, we add AND and OR to the "TRUE FALSE" token type as well as add them to the expression list.

File name | Code

Parser.grm
```
%token <(int*int)> TRUE FALSE AND OR
```

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

Moving on to the type checker, we make sure that we can properly check the used types, when using AND and OR.

```
TypeChecker.sml
```

```
| In.And (e1, e2, pos)
  => let val (t1, e1') = checkExp ftab vtab e1
         val (t2, e2') = checkExp ftab vtab e2
     in case (t1 = t2, t1) of
             (false, _) => raise Error
                           ("And cannot take "^ ppType t1 ^
                            "and "^ppType t2, pos)
           | (true, Array _) =>
             raise Error ("And cannot oporate on arrays", pos)
           | _ => (Bool, Out.And (e1', e2', pos))
    end
| In.Or (e1, e2, pos)
  => let val (t1, e1') = checkExp ftab vtab e1
         val (t2, e2') = checkExp ftab vtab e2
     in case (t1 = t2, t1) of
             (false, _) => raise Error
                           ("Or cannot take "^ ppType t1 ^
                            "and "^ppType t2, pos)
           | (true, Array _) =>
             raise Error ("Or cannot operate on arrays", pos)
           | _ => (Bool, Out.Or (e1', e2', pos))
     end
```

We go ahead and implement them in our interpreter.

```
Interpreter.sml
```

```
| evalExp (And (e1, e2, pos), vtab, ftab) =
      let val res1 = evalExp(e1, vtab, ftab)
      in  case res1 of
            (BoolVal true) => evalExp(e2, vtab, ftab)
          | (BoolVal false) => BoolVal false
          | _ => raise Fail "Arguments to AND is not of type bool"
      end

| evalExp (Or (e1, e2, pos), vtab, ftab) =
      let val res1 = evalExp(e1, vtab, ftab)
      in  case res1 of
            (BoolVal false) => evalExp(e2, vtab, ftab)
          | (BoolVal true)  => BoolVal true
          | _ => raise Fail "Arguments to AND is not of type bool"
      end
```

Finally we add the `AND` and `OR` to the MIPS code generator.

`CodeGen.sml`

```
  | And (e1, e2, pos) =>
    let val t1 = newName "and_L"
        val t2 = newName "and_R"
        val code1 = compileExp e1 vtable t1
        val code2 = compileExp e2 vtable t2
        val finish = newName "finish"
    in  code1 @
        [ Mips.LI (place,"0")
        , Mips.BEQ (t1, "0", finish) ] @
        code2 @
        [ Mips.BEQ (t2, "0", finish)
        , Mips.LI (place, "1")
        , Mips.LABEL finish ]
    end

  | Or (e1, e2, pos) =>
    let val t1 = newName "or_L"
        val t2 = newName "or_R"
        val code1 = compileExp e1 vtable t1
        val code2 = compileExp e2 vtable t2
        val finish = newName "finish"
    in  code1 @
        [ Mips.LI (place,"1")
        , Mips.BNE (t1, "0", finish) ] @
        code2 @
        [ Mips.BNE (t2, "0", finish)
        , Mips.LI (place, "0")
        , Mips.LABEL finish ]
    end
```

2.1.4 `not` and `negate`

Lastly for this task, we will implement `not` and `negate`. We made test files for `not`, but for `negate`, these are already given to us.

File name	Code
`not.fo`	`fun bool notFun(bool t) = not t`
	`fun bool main() =`
	`    let a = read(bool) in`
	`    let b = read(bool) in`
	`    let c = write(notFun(a)) in`
	`    write(notFun(b))`
`not.in`	`1`
	`0`
`not.out`	`falsetrue`

After creating the test files, we move on to the lexer

File name	Code
`Lexer.lex`	`| "not" => Parser.NOT pos`
`Lexer.lex`	`| ‘~‘ { Parser.NEGATE (getPos lexbuf) }`

In the parser, we add `not` and `negate` to the "`TRUE FALSE AND OR`" token type. We make them non-associative. And lastly, we add the two expressions to the expression list.

File name	Code
`Parser.grm`	`%token <(int*int)> TRUE FALSE AND OR NOT NEGATE`
`Parser.grm`	`%nonassoc NOT`
	`%nonassoc NEGATE`
`Parser.grm`	`| NOT Exp    { NOT    ($2, $1) }`
	`| NEGATE Exp { NEGATE ($2, $1) }`

Moving on to the type checker, we make sure that we can properly check the used types, when using `not` and `negate`.

`TypeChecker.sml`

```
| In.Not (e, pos)
  => let val (t1, e') = checkExp ftab vtab e
     in  case (t1) of
           (Bool) => (Bool, Out.Not (e', pos))
         | _ => raise Error ("Invalid Argument type", pos)
     end

| In.Negate (e, pos)
  => let val (t1, e') = checkExp ftab vtab e
     in  case (t1) of
           (Int) => (Int, Out.Negate (e', pos))
         | _ => raise Error ("Invalid Argument type", pos)
     end
```

We go ahead and implement them in our interpreter.

`Interpreter.sml`

```
| evalExp ( Not(e, pos), vtab, ftab ) =
      let val res = evalExp(e, vtab, ftab)
      in  case (res) of
            (BoolVal n) => BoolVal (if n then false else true)
          | _ => invalidOperand "NOT on non-boolean args: "
                   Bool res pos
      end

| evalExp ( Negate(e, pos), vtab, ftab ) =
      let val res = evalExp(e, vtab, ftab)
      in  case (res) of
            (IntVal n) => IntVal (n * (~1))
          | _ => invalidOperand "Negate on non-integer args: "
                   Int res pos
      end
```

Finally we add the expressions to the MIPS code generator

CodeGen.sml

`| Not (e’, pos) =>`
 `     let val t1 = newName "bool"`
 `         val code = compileExp e’ vtable t1`
 `     in  code @`
 `         [Mips.XORI (place, t1, "1")]`
 `     end`
 	
 `| Negate (e’, pos) =>`
 `        let val t1 = newName "negate"`
 `            val code = compileExp e’ vtable t1`
 `        in  code @ [Mips.SUB (place, "0", t1)]`
 `        end`

2.2 Task 2

This task consists of the compiler implementation of `iota`, `map` and `reduce`. `iota` was quite straight forward, as the hints in task 2 guides you through the whole thing; step-by-step. So we will skip explaining the implementation of `iota` and go straight to `map` and `reduce`.

2.2.1 `map` and `reduce`

Just like in task one, we need to extend the lexer, parser, type checker, interpreter and the code generator to accept the `map` and `reduce` operators.

Test files for `map` and `reduce` are already given.

We can now begin with the top of the compiler structure, which is the lexer.

Lexer.lex

`| "map"    => Parser.MAP pos`
 `| "reduce" => Parser.REDUCE pos`

When parsing, we have to keep in mind, that `map` and `reduce` both take functions as the

first argument. We make sure that we can parse function names, anonymous functions. We add `MAP` and `REDUCE` in the token type definition and to the expression list.

File name	Code
Parser.grm	`%token <(int*int)> IOTA MAP`
Parser.grm	`%type <Fasto.UnknownTypes.FunArg> FunArg`

Parser.grm	`| MAP LPAR FunArg COMMA Exp RPAR`
	 `      { Map ($3, $5, (), (), $1) }`
	 `| REDUCE LPAR FunArg COMMA Exp COMMA Exp RPAR`
	 `      { Reduce ($3, $5, $7, (), $1) }`

Moving on to the type checker, we make sure that we can properly check the types that the `map` and `reduce` functions use, extending the `checkExp` function with the proper cases.

```
TypeChecker.sml
```

```
  | In.Map (f, arr_exp, _, _, pos)
    => let val (arr_exp_tp, decvar) = checkExp ftab vtab arr_exp
           val (fnew, f_returntp, f_argument) = checkFunArg(f, vtab,
                       ftab, pos)
           val arr_eltp = case arr_exp_tp of
                              Array t => t
                            | _ => raise Error
                                   ("Map: wrong type of array exp" ,pos)
           val f_argtp =  case f_argument of
                              [tp] => tp
                            | _ => raise Error
                                   ("Map Wrong argument fn type ", pos)
           in if arr_eltp = f_argtp
              then (Array f_returntp, Out.Map (fnew, decvar, arr_eltp,
                                               f_returntp, pos))
              else raise Error ("Map: wrong argument type ", pos)
           end

  | In.Reduce (f, n_exp, arr_exp, _, pos)
    => let val (fnew, f_returntp, f_argument) = checkFunArg(f, vtab,
                ftab, pos)
           val (e_type, n_exp_dec) = checkExp ftab vtab n_exp
           val (arr_exp_tp, decvar) = checkExp ftab vtab arr_exp
           val arr_eltp = case arr_exp_tp of
                              Array t => t
                            | _ => raise Error
                                   ("Reduce: wrong type of array exp" ,pos)
           val f_argtp =  case f_argument of
                              fa::fas => fa(*more*)
                            | _ => raise Error
                                   ("Reduce: Wrong argument fn type ", pos)
           in if e_type = f_argtp andalso arr_eltp = f_argtp
              then (f_returntp, Out.Reduce (fnew, n_exp_dec, decvar,
                    f_returntp, pos))
              else raise Error ("Reduce: Wrong argument type " ^
                                  ppType e_type, pos)
           end
```

Next stop is the interpreter, where we extend the `evalExp` function to be able to interpret the `map` and `reduce` functions.

```
Interpreter.sml
```

```
| evalExp ( Map (farg, arrexp, _, _, pos), vtab, ftab ) =
      let val expression = evalExp(arrexp, vtab, ftab)
          val rtp = rtpFunArg(farg, ftab, pos)
          val f = (fn x => evalFunArg(farg, vtab, ftab, pos, [x]))
      in  case expression of
          ArrayVal (ls, tpvar) => ArrayVal (map (f) (ls), rtp)
        | _ => raise Error ("Argument need to be an ArrayVal", pos)
      end

| evalExp ( Reduce (farg, ne, arrexp, tp, pos), vtab, ftab ) =
      let val expression = evalExp(arrexp, vtab, ftab)
          val neut_exp   = evalExp(ne, vtab, ftab)
          val rtp = rtpFunArg(farg, ftab, pos)
          val f = (fn (x, y) => evalFunArg(farg, vtab, ftab, pos, [x, y]))
      in  case expression of
          ArrayVal (ls, tpvar) => (foldl (f) (neut_exp) (ls))
        | _ => raise Error ("Argument need to be an ArrayVal", pos)
      end
```

Finally the `compileExp` function in the MIPS code generator can be extended with the `map` and `reduce` functions. See appendix A and B for these.

2.3 Task 3

2.3.1 Copy/Constant Propagation

In this sub-task, we add code to the `copyConstPropFoldExp` function, which will produce optimized expression when this is possible. First part is to take care of optimizing `let` expressions recursively and the second part is to replace expressions with reduced expression when this is possible. E.g. `a*1` will always return `a` and hence we replace the expression with `a`. It's important to note that `a*0` can not be replaced with `0` as the expression has side effects.

We have to fill in 5 blanks, but note, that the first two blanks were actually already filled in the given code. For completion, we've included them here.

```
Blanks  Code
(1)     Var (name, pos) =>
        (case SymTab.lookup name vtable of
              SOME (VarProp newname) => Var (newname, pos)
            | SOME (ConstProp value) => Constant (value, pos)
            | _                      => Var (name, pos))

(2)     | Index (name, e, t, pos) =>
          (case SymTab.lookup name vtable of
                SOME (VarProp newname) =>
                  Index (newname, copyConstPropFoldExp vtable e,
                         t, pos)
              | _ =>
                  Index (name, copyConstPropFoldExp vtable e,
                         t, pos))

(3)     Var (varname, _) =>
        let val vtable2 = SymTab.bind name (VarProp varname) vtable
            val body2 = copyConstPropFoldExp vtable2 body
        in
            Let (Dec (name, e', decpos), body2, pos)
        end

(4)     | Constant (value, _) =>
          let val vtable2 = SymTab.bind name (ConstProp value) vtable
              val body2 = copyConstPropFoldExp vtable2 body
          in
              Let (Dec (name, e', decpos), body2, pos)
          end

(5)     | Let (Dec bindee, inner_body, inner_pos) =>
          copyConstPropFoldExp vtable (Let (Dec bindee,
              Let (Dec (name, inner_body, inner_pos), body, pos), pos))
```

2.3.2 Constant Folding

In this sub-task the implement constant folding for {`*, /, &&, ||, ==, not`}. Do note, that in the given code; {`/, ||, ==, not`} were already implemented. We have included them below for the sake of completion.

```
Blanks  Code

'*'     | Times (e1, e2, pos) =>
          let val e1' = copyConstPropFoldExp vtable e1
              val e2' = copyConstPropFoldExp vtable e2
          in case (e1', e2') of
                  (Constant (IntVal x, _), Constant (IntVal y, _)) =>
                   Constant (IntVal (x*y), pos)
                | (Constant (IntVal 1, _), _) => e2'
                | (_, Constant (IntVal 1, _)) => e1'
                | _ => Times (e1', e2', pos)
          end

'/'     | Divide (e1, e2, pos) =>
          let val e1' = copyConstPropFoldExp vtable e1
              val e2' = copyConstPropFoldExp vtable e2
          in case (e1', e2') of
                  (Constant (IntVal x, _), Constant (IntVal y, _)) =>
                  (Constant (IntVal (Int.quot (x,y)), pos)
                   handle Div => Divide (e1', e2', pos))
                | _ => Divide (e1', e2', pos)
          end

'&&'    | And (e1, e2, pos) =>
          let val e1' = copyConstPropFoldExp vtable e1
              val e2' = copyConstPropFoldExp vtable e2
          in case (e1', e2') of
                  (Constant (BoolVal x, _), Constant (BoolVal y, _)) =>
                   Constant (BoolVal (x andalso y), pos)
                | _ => And (e1', e2', pos)
          end

'||'    | Or (e1, e2, pos) =>
          let val e1' = copyConstPropFoldExp vtable e1
              val e2' = copyConstPropFoldExp vtable e2
          in case (e1', e2') of
                  (Constant (BoolVal a, _), Constant (BoolVal b, _)) =>
                   Constant (BoolVal (a orelse b), pos)
                | _ => Or (e1', e2', pos)
          end

'=='    | Equal (e1, e2, pos) =>
          let val e1' = copyConstPropFoldExp vtable e1
              val e2' = copyConstPropFoldExp vtable e2
          in case (e1', e2') of
                  (Constant (v1, _), Constant (v2, _)) =>
                   Constant (BoolVal (v1 = v2), pos)
                | _ => if e1' = e2'
                       then Constant (BoolVal true, pos)
                       else Equal (e1', e2', pos)
          end
```

Appendix A map and reduce

```
CodeGen.sml
```

```
| Map (farg, arr_exp, elem_type, ret_type, pos) =>
      let val arr_reg = newName "arr_reg"
          val len_reg = newName "len_reg"
          val res_reg = newName "res_reg"
          val i_reg   = newName "i_reg"
          val tmp_reg = newName "tmp_reg"

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

          val code1 = compileExp arr_exp vtable arr_reg
          val code2 = [Mips.LW (len_reg, arr_reg, "0")]

          val init_regs = [ Mips.ADDI (arr_reg, arr_reg, "4")
                          , Mips.ADDI (res_reg, place, "4")
                          , Mips.MOVE (i_reg, "0") ]

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

          val loop_map_load = case getElemSize elem_type of
                 One => Mips.LB (tmp_reg, arr_reg, "0") ::
                        applyFunArg (farg, [tmp_reg], vtable, tmp_reg, pos) @
                        [Mips.ADDI(arr_reg, arr_reg, "1")]
               | Four => Mips.LW (tmp_reg, arr_reg, "0") ::
                        applyFunArg (farg, [tmp_reg], vtable, tmp_reg, pos) @
                        [Mips.ADDI(arr_reg, arr_reg, "4")]

          val loop_map_store = case getElemSize ret_type of
                                   One  => [ Mips.SB (tmp_reg, res_reg, "0")
                                           , Mips.ADDI(res_reg, res_reg, "1")]
                                 | Four => [ Mips.SB (tmp_reg, res_reg, "0")
                                           , Mips.ADDI(res_reg, res_reg, "4")]

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

      in   code1
         @ code2
         @ dynalloc (len_reg, place, ret_type)
         @ init_regs
         @ loop_header
         @ loop_map_load
         @ loop_map_store
         @ loop_footer
      end
```

Appendix B map and reduce

```
CodeGen.sml
```

```
(* reduce(f, acc, {x1, x2, ...}) = f(..., f(x2, f(x1, acc))) *)
| Reduce (binop, ne_exp, arr_exp, tp, pos) =>
        let val ne_reg  = newName "ne_reg"
            val arr_reg = newName "arr_reg"
            val len_reg = newName "len_reg"
            val res_reg = newName "res_reg"
            val i_reg   = newName "i_reg"
            val tmp_reg = newName "tmp_reg"

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

            val ne_code = compileExp ne_exp vtable ne_reg
            val arr_code = compileExp arr_exp vtable arr_reg
            val len_code = [Mips.LW (len_reg, arr_reg, "0")]

            val init_regs = [ Mips.ADDI (arr_reg, arr_reg, "4")
                            , Mips.MOVE (i_reg, "0")
                            , Mips.MOVE (place, ne_reg) ]

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

            val loop_reduce = case getElemSize tp of
                   One => Mips.LB (tmp_reg, arr_reg, "0") ::
                          applyFunArg (binop, [place, tmp_reg], vtable,
                                       place, pos) @
                          [Mips.ADDI(arr_reg, arr_reg, "1")]
                 | Four => Mips.LW (tmp_reg, arr_reg, "0") ::
                          applyFunArg (binop, [place, tmp_reg], vtable,
                                       place, pos) @
                          [Mips.ADDI(arr_reg, arr_reg, "4")]

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

        in   ne_code
           @ arr_code
           @ len_code
           @ init_regs
           @ loop_header
           @ loop_reduce
           @ loop_footer
        end
```