Compilers
Group Assignment

Martin Bech Hansen - pwl435
Signe S. Jensen - smg399
Thorkil Simon Kowalski Værge - wng750

22/12-14

Indhold

1 Introduktion

This report covers extensions for the Fasto compiler as covered in the group assignment. We will go into detail with the parts of the tasks we found interesting and/or challenging and describe our solutions to them as well as including the relevant code. We'll also briefly discuss the tests we've performed.

2 Task 1 - Extra operators

As described in the assignment we've extended the Fasto compiler with the new functionality listed below:

From Lexer.lex:
We've added "not", "true"and "false"to the listed keywords:

```
1  fun keyword (s, pos) =
2      case s of
3  ...
4        | "not"          => Parser.NOT pos
5        | "true"         => Parser.TRUE pos
6        | "false"        => Parser.FALSE pos
```

And added Times, Div, And, Or and Negate to the parsed tokens:

```
1  rule Token = parse
2  ...
3    | `*`                { Parser.TIMES   (getPos lexbuf) }
4    | `/`                { Parser.DIV     (getPos lexbuf) }
5    | `&``&`             { Parser.AND     (getPos lexbuf) }
6    | `|``|`             { Parser.OR      (getPos lexbuf) }
7    | `~`                { Parser.NEGATE  (getPos lexbuf) }
```

From Parser.grm:
We've added a case for each of the new operators and the two constants in the Expression declaration.

```
1  Exp :
2  ...
3   | TRUE            { Constant (BoolVal (true), $1) }
4   | FALSE           { Constant (BoolVal (false), $1) }
5  ...
6   | Exp AND   Exp   { And ($1, $3, $2) }
7   | Exp OR    Exp   { Or ($1, $3, $2) }
8   | NOT       Exp   { Not ($2, $1) }
9   | NEGATE    Exp   { Negate ($2, $1) }
10  | Exp TIMES Exp   { Times ($1, $3, $2) }
11  | Exp DIV   Exp   { Divide ($1, $3, $2) }
```

2.1 Precedence

To ensure the operators have the precedence specified in the assignment text we've listed each of them in the precedence hierarchy in Parser.grm as follows:

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

This way Negate will bind the strongest, And will bind stronger than Or, Not will bind weaker than the logical comparisons and Times and Div will bind stronger than Plus and Minus.

2.2 And and Or

The most interesting of the operations here were the implementations of And and Or done in Interpreter.sml and CodeGen.sml which had to be done with short-circuiting:
Example of And from 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
      end
```

Example of And from CodeGen.sml:

```
| And (e1, e2, pos) =>
      let val thenLabel = newName "andthen"
          val elseLabel = newName "andelse"
          val endLabel = newName "andend"
          val code1 = compileCond e1 vtable thenLabel elseLabel
          val code2 = compileExp e2 vtable place
      in code1 @ [Mips.LABEL thenLabel] @ code2  @
         [ Mips.J endLabel, Mips.LABEL elseLabel, Mips.LI (place
            ,"0"), Mips.LABEL endLabel]
      end
```

As suggested we used compileCond to save jumps.
Or was implemented similarly.

2.3 Negate and Not

We made Negate by compiling the expression and subtracting the resulting integer from 0.

```
| Negate (e, pos) =>
    let val t = newName "Negate_"
```

```
3             val code = compileExp e vtable t
4         in code @ [Mips.SUB (place,"0",t)]
5         end
```

We Not by evaluating the boolean and xoring it with the integer 1.

```
1     | Not (e, pos) =>
2         let val t = newName "not_"
3             val code = compileExp e vtable t
4         in code @ [Mips.XORI (place,t,"1")]
5         end
```

2.4 True and False

True and False were also a bit interesting as they are not operators, but constants.
As Booleans do not exist in Assembly we implemented them using the integers 0 and 1.

```
1   | Constant (BoolVal b, pos) =>
2       (case b of
3           true    => [ Mips.LI (place,"1") ]
4         | false   => [ Mips.LI (place,"0") ]
5       )
```

We also added the appropriate cases for each operator in the type checker and interpreter, in most cases based on the existing functionality from Plus and Minus. See Appendixes Task 1 - TypeChecker.sml and Task 1 - Interpreter.sml for the full implementations.

2.5 Tests

All the tests that came with Fasto runs with the desired result, taking into consideration that we haven't implemented Task 5.
We have also made a series of tests ourselves:
Mult: (mult.fo)
-i: The function returns an int when getting two integers as input. It gives an interpreter error on chars, strings and booleans.
-c: The function returns an int when getting two integers as input. It gives an error when running in Mars with any other input.

Div: (div.fo)
-i: The function returns an int when getting two integers as input, except when dividing with 0 when it returns and error. It gives an interpreter error on chars, strings and booleans.
-c: Returns the value of two integers divided. Returns an error on division by 0 and chars, and strings.

And: (And.fo, AndFail.fo)
-i: Returns the correct boolean values depending on input, and returns an interpreter error on chars, strings.
-c: Returns the true or false depending on the input values(0 or 1), on all other input it returns an error.

AndFail:
-c: Tries to use And on input 000 and 00. Returns an error.

Or: (Or.fo)
-i: Returns the correct boolean values depending on input, and returns an interpreter error on chars, strings.
-c: Returns the true or false depending on the input values(0 or 1), on all other input it returns an error.

AndOr: (AndOr.fo)
-i: The function returns a string containing the text *This is the correct result.* by using And and Or on booleans. It shows that the operators work.
-c: The function returns a string containing the text *This is the correct result.* by using And and Or on booleans. It shows that the operators work.

Not: (Not.fo)
-i: Returns the correct boolean values depending on input, and returns an interpreter error on chars, strings.
-c: Returns the true on false and false on true and all other integers. It returns an error on chars and strings.

Negate: (neg_simple.fo)
-i -c: Returns the negated integer values.

3 Task 2 - Filter and Scan

We implemented Filter and Scan in the CodeGenerator. For each function we'll show the code here and then give a line by line explanation of our code.

3.1 Filter

Implenteing Filter in CodeGen.sml was fairly straightforward. We used Map as inspiration and simply evaluated the function argument for each list element and only copied the element to the new array if the function call evaluated to true.

```
1  fun compileExp e vtable place = case e of
2  ...
3  | Filter (farg, arr_exp, elem_type, pos) =>
4  let val size_reg = newName "size_reg"
5      val arr_reg  = newName "arr_reg"
6      val elem_reg = newName "elem_reg"
7      val res_reg  = newName "res_reg"
8      val arr_code = compileExp arr_exp vtable arr_reg
9      val get_size = [ Mips.LW (size_reg, arr_reg, "0") ]
10     val addr_reg = newName "addr_reg"
11     val i_reg = newName "i_reg"
12     val new_counter = newName "zc"
13     val init_regs = [ Mips.ADDI (addr_reg, place, "4")
14                     , Mips.MOVE (i_reg, "0")
15                     , Mips.ADDI (elem_reg, arr_reg, "4") ]
16     val loop_beg = newName "loop_beg"
17     val loop_end = newName "loop_end"
18     val tmp_reg = newName "tmp_reg"
19     val loop_header = [ Mips.LABEL (loop_beg)
20                       , Mips.SUB (tmp_reg, i_reg, size_reg)
21                       , Mips.BGEZ (tmp_reg, loop_end) ]
22     val loop_map0 =
23         let val crlabel = newName "cond_result"
24             val code0 = case getElemSize elem_type of
25                 One => [ Mips.LB(res_reg, elem_reg, "0") ]
26               | Four => [ Mips.LW(res_reg, elem_reg, "0") ]
27             val code1 = applyFunArg(farg, [res_reg], vtable,
                   crlabel, pos)
28             val dontCopyLabel = newName "increment"
29             val copycode = case getElemSize elem_type of
30                 One => [ Mips.SB(res_reg, addr_reg, "0") ]
31                      @ [ Mips.ADDI(addr_reg, addr_reg, "1"),
32                          Mips.ADDI(new_counter,new_counter,"1")]
33               | Four => [ Mips.SW(res_reg, addr_reg, "0") ]
34                      @ [ Mips.ADDI(addr_reg, addr_reg, "4"),
35                          Mips.ADDI(new_counter,new_counter,"1") ]
36             val incrementcode = case getElemSize elem_type of
37                 One => Mips.ADDI (elem_reg, elem_reg, "1")
```

```
38                | Four => Mips.ADDI (elem_reg, elem_reg, "4")
39          in
40              code0
41            @ code1
42            @ [Mips.BEQ (crlabel,"0",dontCopyLabel)]
43            @ copycode
44            @ [Mips.LABEL dontCopyLabel,incrementcode]
45          end
46      val write_new_size =
47          [ Mips.SW (new_counter,place,"0") ]
48      val loop_footer =
49          [ Mips.ADDI (i_reg, i_reg, "1")
50          , Mips.J loop_beg
51          , Mips.LABEL loop_end]
52  in arr_code
53     @ get_size
54     @ dynalloc (size_reg, place, elem_type)
55     @ init_regs
56     @ loop_header
57     @ loop_map0
58     @ loop_footer
59     @ write_new_size
60  end
```

Line 4 through 18 initialize the needed temporary registers, line 8 compiles the array-expression and 9 loads the first word of the compiled array to get its size. *elem_reg* and *addr_reg* are set to point to the first element in each of the two arrays.
Line 19-21 contain the loop header, a label and a conditional branch testing whether our counter has reached the size of the array.
Line 22-45 contain the body of the loop itself, first loading the value from the old array (24-26) then calling our function argument with the loaded value (27). Line 42 then makes sure we only copy the appropriate elements by checking the returned boolean value and skipping the copying code, which also increments a size-counter for the new array (29-35), when needed and then incrementing the pointer to the old array (36-38).
The loop footer (48-51) increments the loop counter and jumps back to the condition of the loop. Lastly the size of the new array is written in *place*, which is the first word of the new array.

3.2 Scan

Again we used Map for inspiration. We first tried using Reduce, as suggested in the comments, but since we already had a good understanding of Map it worked out better when using this.

```
1  fun compileExp e vtable place = case e of
2  ...
3   | Scan (farg, acc_exp, arr_exp, tp, pos) =>
4  let val size_reg = newName "size_reg"
5      val size_reg = newName "size_reg"
6      val arr_reg  = newName "arr_reg"
7      val elem_reg = newName "elem_reg"
```

```
8       val res_reg  = newName "res_reg"
9       val e_reg    = newName "e_reg"
10      val arr_code = compileExp arr_exp vtable arr_reg
11      val acc_code = compileExp acc_exp vtable e_reg
12      val get_size = [ Mips.LW (size_reg, arr_reg, "0"),
13                       Mips.ADDI(size_reg,size_reg,"1") ]
14      val addr_reg = newName "addr_reg"
15      val i_reg = newName "i_reg"
16      val init_regs =
17                  let
18                val first_elem = case getElemSize tp of
19                    One =>  [ Mips.SB (e_reg, addr_reg, "0"),
20                              Mips.ADDI (addr_reg, addr_reg, "1") ]
21                  | Four => [ Mips.SW (e_reg, addr_reg, "0"),
22                              Mips.ADDI (addr_reg, addr_reg, "4") ]
23              in
24                  [ Mips.ADDI (addr_reg, place, "4") ]
25                @ first_elem
26                @ [ Mips.MOVE (i_reg, "0")
27                  , Mips.ADDI (elem_reg, arr_reg, "4") ]
28              end
29      val loop_beg = newName "loop_beg"
30      val loop_end = newName "loop_end"
31      val tmp_reg = newName "tmp_reg"
32      val loop_header = [ Mips.LABEL (loop_beg)
33                        , Mips.SUB (tmp_reg, i_reg, size_reg)
34                        , Mips.BGEZ (tmp_reg, loop_end) ]
35      val loop_scan0 =
36          case getElemSize tp of
37            One =>  Mips.LB   (tmp_reg, elem_reg, "0")
38                    :: applyFunArg(farg, [e_reg, tmp_reg], vtable,
                          res_reg, pos)
39                    @ [ Mips.MOVE (e_reg, res_reg) ]
40                    @ [ Mips.ADDI (elem_reg, elem_reg, "1") ]
41          | Four => Mips.LW   (tmp_reg, elem_reg, "0")
42                    :: applyFunArg(farg, [e_reg, tmp_reg], vtable,
                          res_reg, pos)
43                    @ [ Mips.MOVE (e_reg, res_reg) ]
44                    @ [ Mips.ADDI (elem_reg, elem_reg, "4") ]
45      val loop_scan1 =
46          case getElemSize tp of
47              One => [ Mips.SB (res_reg, addr_reg, "0") ]
48            | Four => [ Mips.SW (res_reg, addr_reg, "0") ]
49      val loop_footer =
50          [ Mips.ADDI (addr_reg, addr_reg,
51                       makeConst (elemSizeToInt (getElemSize tp)))
52          , Mips.ADDI (i_reg, i_reg, "1")
53          , Mips.J loop_beg
54          , Mips.LABEL loop_end
```

```
55          ]
56  in arr_code
57     @ acc_code
58     @ get_size
59     @ dynalloc (size_reg, place, tp)
60     @ init_regs
61     @ loop_header
62     @ loop_scan0
63     @ loop_scan1
64     @ loop_footer
65  end
```

FIXME PUT AN EXPLANATION OF SCAN HERE.

3.3 Tests

FIXME Put in some tests here. I don't know what they are, but they don't need to be complete.

4 Task 3 - Lambda-expressions

In the type checker we added a case for anonymous functions in checkFunArg (line 368). As suggested in the comments we construct a FunDec, pass it to checkFunWithVtable and then construct a Lambda from the result. The rest of the case is similar to that of normal function arguments.

```
1  and checkFunArg (In.FunName fname, vtab, ftab, pos) =
2  ...
3    | checkFunArg (In.Lambda (ret_type, params, exp, funpos),
4                   vtab, ftab, pos) =
5        let val Out.FunDec (fname, ret_type, args, body, pos) =
6                checkFunWithVtable (In.FunDec ("anon", ret_type,
                    params, exp, funpos),vtab, ftab, pos)
7            val arg_types = map (fn (Param (_, ty)) => ty) args
8            in (Out.Lambda (ret_type, args, body, funpos),
9                ret_type, arg_types)
10       end
```

The interesting part happens in line 6 where we construct a function declaration with a bogus name and call CheckFunWithVtable, line 7 where we strip the parameter list to get their types and line 8 where we construct an Out.Lambda and return it along with return and argument types.

In the Interpreter we added a case to evalFunArg (line 514) and like in the type checker we construct a function declaration with a bogus name, and then pass it to callFunWithVtable.

```
1  and evalFunArg (FunName fid, vtab, ftab, callpos) =
2  ...
3    | evalFunArg (Lambda (ret_type, args, body, funpos),
4                  vtab, ftab, callpos) =
5              (fn aargs =>
6                 callFunWithVtable (
7                   FunDec ("anon", ret_type, args, body, funpos),
8                   aargs, vtab, ftab, callpos), ret_type)
```

Again the interesting part happens in line 6 where we call callFunWithVtable and pass it a constructed function declaration and the existing vtable.

In CodeGen.sml our extra case handles adding the actual arguments to the existing vtable and then inlining the code by compiling it in place with the new modified vtable and moving the resulting value to *place* where it belongs.

```
1  and applyFunArg (FunName s, ... ) : Mips.Prog =
2  ...
3    | applyFunArg (Lambda (ret_type, args, body, funpos),
4                   aargs, vtab, place, callpos) : Mips.Prog =
5       let val fun_res = newName "fun_res" (* for the result *)
6           val arg_names = map (fn Param (n,t) => n) args
7           val zipped_list = zip arg_names aargs
8           val argtab = SymTab.fromList zipped_list
```

```
9           val vtab' = SymTab.combine vtab argtab
10          val fun_code = compileExp body vtab' fun_res
11      in fun_code @ [Mips.MOVE(place, fun_res)]
12      end
```

We have added a small helper-function named zip to easily combine the argument names with the passed argument values in order to combine them with the existing vtable. It simply takes a list of formal argument names, matches it up with a list of symbolic registers containing the actual arguments and returns a list of tuples with the pairs.

In line 5 we make a new register for the function result. In line 6-9 we make the list of parameters and add them to the existing vtable. In line 10 we compile the actual body of the function and let i place the result in fun_res and in line 11 we call the body-compilation and then move the result into *place*.

4.1 Tests

Lamda-opg.fo
We have copied this test from the assignment text, though we have made a few adjustments:

```
1 fun [int] main() =
2         let n = read(int) in
3         let a = map(fn int (int i) => read(int), iota(n))
4         in let x = read(int)
5         in let b = map(fn int (int y) => write(x + y), a)
6         in (b)
```

The original program attempted to pass Write a list of integers, which is not possible, so we have moved the call to write into the anonymous function in line 5 and changed the return type of main from [char] to [int] to accommodate the new return value.
The test runs as expected.

5 Task 4 - ConstCopyPropFold

We have edited two cases to the switch-case of

```
1 fun copyConstPropFoldExp vtable e =
```

to implement the intended behaviour of constant folding and copy propagation when it comes to variables and values defined in let-clauses.
In the case of Var we take a look in the vtable and if we find a constant of propegatee we return that, else we return the variable as is.

```
1  | Var (name, pos) => ( case SymTab.lookup name vtable of
2                         SOME (ConstProp x)         => Constant (x,
                             pos)
3                       | SOME (VarProp proppedVar) => Var (
                          proppedVar,pos)
4                       | NONE                       => Var (name, pos
                          )
5        )
```

In the case of Let we call copyConstPropFoldExp on the expression recursively and then add it to our vtable if it's foldable (can be reduced to a constant or variable). At first we simply returned the current vtable if the expression was not foldable, but now we remove any older bindings to the same name to prevent shadowing.

```
1  | Let (Dec (name, e, decpos), body, pos) =>
2    let val e' = copyConstPropFoldExp vtable e
3        val vtable' =  case e' of
4                         Constant (x,_)         => SymTab.bind name (
                             ConstProp x) vtable
5                       | Var (x,_)              => SymTab.bind name (
                          VarProp x) vtable
6                       | _                      => SymTab.remove name
                            vtable
```

5.1 Shadowing

An example of shadowing:

```
1    let x = 5 in
2    let x = f(x) in
3    x * x
```

In this case x will be bound to our vtable as 5, then x = f(x) (if it cannot be folded) will remove the old binding of x and x*x cannot be folded. If we had not added the call to remove this expression would be folded to the constant 25, as x*x would find the first binding of x.

We have not found a solution to the example of shadowing shown in the assignment text.

5.2 Tests

FIXME Add the test

6 Appendices

6.1 Task 1 - TypeChecker.sml

```
    | In.And (e1, e2, pos)
      => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos,
         Bool, e1, e2)
         in (Bool,
             Out.And (e1_dec, e2_dec, pos))
         end
    | In.Or (e1, e2, pos)
      => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos,
         Bool, e1, e2)
         in (Bool,
             Out.Or (e1_dec, e2_dec, pos))
         end
    | In.Mult (e1, e2, pos)
      => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos,
         Int, e1, e2)
         in (Int,
             Out.Mult (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
    | In.Negate (e1, pos)
      => let val (_, e1_dec) = checkUnOp ftab vtab (pos, Int, e1)
         in (Int,
             Out.Negate (e1_dec, pos))
         end
    | In.Not (e1, pos)
      => let val (_, e1_dec) = checkUnOp ftab vtab (pos, Bool, e1)
         in (Bool,
             Out.Not (e1_dec, pos))
```

6.2 Task 1 - Interpreter.sml

```
  | evalExp ( Not(e, pos), vtab, ftab ) =
        let val res   = evalExp(e, vtab, ftab)
        in  evalUnopBool(not, res, pos)
        end
```

```
5   | evalExp ( And(e1, e2,  pos), vtab, ftab ) =
6         let val res1 = evalExp(e1, vtab, ftab)
7         in case res1 of
8               BoolVal true  => evalExp(e2, vtab, ftab)
9             | BoolVal false => BoolVal false
10        end
11  | evalExp ( Or(e1, e2,  pos), vtab, ftab ) =
12        let val res1 = evalExp(e1, vtab, ftab)
13        in case res1 of
14              BoolVal false  => evalExp(e2, vtab, ftab)
15            | BoolVal true => BoolVal true
16        end
17  | evalExp ( Negate(e, pos), vtab, ftab ) =
18        let val res   = evalExp(e, vtab, ftab)
19        in  evalUnopNum(op ~, res, pos)
20        end
21  | evalExp ( Mult(e1, e2, pos), vtab, ftab ) =
22        let val res1   = evalExp(e1, vtab, ftab)
23            val res2   = evalExp(e2, vtab, ftab)
24        in  evalBinopNum(op *, res1, res2, pos)
25        end
26  | evalExp ( Divide(e1, e2, pos), vtab, ftab ) =
27        let val res1   = evalExp(e1, vtab, ftab)
28            val res2   = evalExp(e2, vtab, ftab)
29        in  evalBinopNum(op Int.quot, res1, res2, pos)
30        end
```

6.3 Scan - CodeGen.sml

```
1   | Scan (farg, acc_exp, arr_exp, tp, pos) =>
2       let val size_reg = newName "size_reg" (* size of input array
            *)
3           val size_reg = newName "size_reg" (* size of output array
                *)
4           val arr_reg  = newName "arr_reg" (* address of new array
                *)
5           val elem_reg = newName "elem_reg" (* address of single
                element *)
6           val res_reg = newName "res_reg" (* vrdi fra input arr og
                resultat af funktionen *)
7           val e_reg = newName "e_reg" (* vores udregnede vrdi der
                skal bruges i nste iteration*)
8           val arr_code = compileExp arr_exp vtable arr_reg
9           val acc_code = compileExp acc_exp vtable e_reg      (*
                her udregner vi vores frste e *)
10
11          val get_size = [ Mips.LW (size_reg, arr_reg, "0"),
```

```
12                      Mips.ADDI(size_reg,size_reg,"1") ] (*
                           fordi det nye array er et element
                           lngere. *)
13
14          val addr_reg = newName "addr_reg" (* address of element
                in new array *)
15          val i_reg = newName "i_reg"
16          val init_regs = let
17                          val first_elem = case getElemSize tp
                                of
18                           One =>  [ Mips.SB (e_reg,
                                addr_reg, "0"),
19                                    Mips.ADDI (addr_reg,
                                       addr_reg, "1") ]
20                         | Four => [ Mips.SW (e_reg,
                             addr_reg, "0"),
21                                    Mips.ADDI (addr_reg,
                                       addr_reg, "4") ]
22                      in
23                      [ Mips.ADDI (addr_reg, place, "4") ]
24                      @ first_elem
25                      @ [ Mips.MOVE (i_reg, "0")
26                      , Mips.ADDI (elem_reg, arr_reg, "4") ]
27                      end
28          val loop_beg = newName "loop_beg"
29          val loop_end = newName "loop_end"
30          val tmp_reg = newName "tmp_reg"
31          val loop_header = [ Mips.LABEL (loop_beg)
32                            , Mips.SUB (tmp_reg, i_reg, size_reg)
33                            , Mips.BGEZ (tmp_reg, loop_end) ]
34
35          val loop_scan0 =
36              case getElemSize tp of
37                  One =>  Mips.LB   (tmp_reg, elem_reg, "0")
38                          :: applyFunArg(farg, [e_reg, tmp_reg],
                               vtable, res_reg, pos)
39                          @ [ Mips.MOVE (e_reg, res_reg) ]
40                          @ [ Mips.ADDI (elem_reg, elem_reg, "1") ]
41                | Four => Mips.LW   (tmp_reg, elem_reg, "0")
42                          :: applyFunArg(farg, [e_reg, tmp_reg],
                               vtable, res_reg, pos)
43                          @ [ Mips.MOVE (e_reg, res_reg) ]
44                          @ [ Mips.ADDI (elem_reg, elem_reg, "4") ]
45
46          val loop_scan1 =
47              case getElemSize tp of
48                  One => [ Mips.SB (res_reg, addr_reg, "0") ]
49                | Four => [ Mips.SW (res_reg, addr_reg, "0") ]
50
```

```
51          val loop_footer =
52              [ Mips.ADDI (addr_reg, addr_reg,
53                           makeConst (elemSizeToInt (getElemSize tp
                                 )))
54              , Mips.ADDI (i_reg, i_reg, "1")
55              , Mips.J loop_beg
56              , Mips.LABEL loop_end
57              ]
58      in arr_code
59         @ acc_code                        (* compile vores frste e-
              element *)
60         @ get_size
61         @ dynalloc (size_reg, place, tp)
62         @ init_regs
63         @ loop_header
64         @ loop_scan0
65         @ loop_scan1
66         @ loop_footer
67      end
```