

Compilers

Group Assignment

Martin Bech Hansen - pwl435

Signe S. Jensen - smg399

Thorkil Simon Kowalski Værge - wng750

22/12-14

Indhold

| | | |
|----------|------------------------------------|-----------|
| 1 | Introduktion | 3 |
| 2 | Task 1 - Extra operators | 3 |
| 2.1 | Precedence | 3 |
| 2.2 | And and Or | 4 |
| 2.3 | Negate and Not | 4 |
| 2.4 | True and False | 5 |
| 2.5 | Tests | 5 |
| 3 | Task 2 - Filter and Scan | 7 |
| 3.1 | Filter | 7 |
| 3.2 | Scan | 8 |
| 3.3 | Tests | 10 |
| 4 | Task 3 - Lambda-expressions | 11 |
| 4.1 | Tests | 12 |
| 5 | Task 4 - ConstCopyPropFold | 13 |
| 5.1 | Shadowing | 13 |
| 5.2 | Tests | 13 |
| 6 | Appendices | 15 |
| 6.1 | Task 1 - TypeChecker.sml | 15 |
| 6.2 | Task 1 - Interpreter.sml | 15 |

1 Introduction

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) = case s of
2   ...
3   | "not"           => Parser.NOT pos
4   | "true"          => Parser.TRUE pos
5   | "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:

```

1 %nonassoc ifprec letprec
2 %left DEQ LTH
3 %left PLUS MINUS
4 %left TIMES DIV
5 %nonassoc NOT
6 %left OR
7 %left AND
8 %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**:

```

1 | evalExp ( And(e1, e2, pos), vtab, ftab ) =
2   let val res1 = evalExp(e1, vtab, ftab)
3   in case res1 of
4       BoolVal true  => evalExp(e2, vtab, ftab)
5       | BoolVal false => BoolVal false
6   end

```

Example of *And* from **CodeGen.sml**:

```

1 | And (e1, e2, pos) =>
2   let val thenLabel = newName "andthen"
3       val elseLabel = newName "andelse"
4       val endLabel = newName "andend"
5       val code1 = compileCond e1 vtable thenLabel elseLabel
6       val code2 = compileExp e2 vtable place
7   in code1 @ [Mips.LABEL thenLabel] @ code2 @
8     [ Mips.J endLabel, Mips.LABEL elseLabel, Mips.LI (place
9       , "0"), Mips.LABEL endLabel]
9   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.

```

1 | Negate (e, pos) =>
2   let val t = newName "Negate_"
3       val code = compileExp e vtable t
4   in code @ [Mips.SUB (place, "0", t)]
5   end

```

We made *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** run with the desired results, 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,
28                                   crlabel, pos)
29            val dontCopyLabel = newName "increment"
30            val copycode = case getElemSize elem_type of
31                            One => [ Mips.SB(res_reg, addr_reg, "0") ]
32                            @ [ Mips.ADDI(addr_reg, addr_reg, "1"),
33                               Mips.ADDI(new_counter, new_counter, "1")]
34                            | Four => [ Mips.SW(res_reg, addr_reg, "0") ]
35                            @ [ Mips.ADDI(addr_reg, addr_reg, "4"),
36                               Mips.ADDI(new_counter, new_counter, "1") ]
37            val incrementcode = case getElemSize elem_type of
38                                  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,
39                               res_reg, pos)
40                 @ [ Mips.MOVE (e_reg, res_reg) ]
41                 @ [ Mips.ADDI (elem_reg, elem_reg, "1") ]
42             | Four => Mips.LW (tmp_reg, elem_reg, "0")
43                 :: applyFunArg(farg, [e_reg, tmp_reg], vtable,
44                               res_reg, pos)
45                 @ [ Mips.MOVE (e_reg, res_reg) ]
46                 @ [ Mips.ADDI (elem_reg, elem_reg, "4") ]
47     val loop_scan1 =
48         case getElemSize tp of
49             One => [ Mips.SB (res_reg, addr_reg, "0") ]
50             | Four => [ Mips.SW (res_reg, addr_reg, "0") ]
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

```

```

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

```

Lines 4 through 31 initialize the necessary registers. Line 10 and 11 compile the passed array and base element expressions. In *init_regs* the base element is copied to the first element of the new array.

The loop header (32-34) is similar to that of Filter. The loop body (35-44) loads the element from the old array, calls the passed function with this and the saved base element, increments the proper registers and saves the function result in both the new array (25-27) and as the new base element value (39).

Lines 55-64 structure the code (and calls *dynalloc* to allocate memory for the new array.)

3.3 Tests

Scan.fo

Runs as expected.

Filter.fo

Runs as expected.

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,
7         params, exp, funpos), vtab, ftab, pos)
8       val arg_types = map (fn (Param (_, ty)) => ty) args
9       in (Out.Lambda (ret_type, args, body, funpos),
10         ret_type, arg_types)
11   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
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 is 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 (
5                       | Var (x,_)          => SymTab.bind name (
6                       | _                  => SymTab.remove name
7                       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

CopyConstProFold.fo

```

1 fun int main() =
2   let a = 3 in
3   let b = a in
4   write (b)

```

Should be (and is) optimized to

```

1 fun int main() = write (3)

```

CopyConstProFold2.fo

Test with function calls.

Runs as expected.

CopyConstProFoldEqual.fo

```

1 fun [char] main () =
2   let a = read(int) in
3   let b = read(int) in
4   if a == b then write("They are equal.") else write ("They are
      not equal.")

```

This test runs as expected, but:

```

1 fun copyConstPropFoldExp vtable e = case e of
2   ...
3   | Equal (e1, e2, pos) =>
4     let val e1' = copyConstPropFoldExp vtable e1
5         val e2' = copyConstPropFoldExp vtable e2
6     in case (e1', e2') of
7       (Constant v1, Constant v2) =>
8         Constant (BoolVal (v1 = v2), pos)
9     | _ => if e1' = e2'
10          then Constant (BoolVal true, pos)
11          else Equal (e1', e2', pos)
12   end

```

It has been brought to Cosmin's attention that checking whether the constant values in this snippet are equal will never evaluate to *True* as the function compares the two constants including their positions, which will always differ.

We have not had time to test the following theory, but seeing that each expression also carries a position, it is apparent that the equality check in line 9 will also never evaluate to *True*, for the same reason.

We also believe that if we, as Cosmin suggests, get rid of the positions in the comparison, another bug will show. Consider the test described above: The expressions **a** and **b** are identical and so they would be considered equal if not for their respective positions, and therefore they would be replaced by a *True* boolean value in line 4 of the test, which is obviously wrong as the user might input two different integers. The same would happen with equal expressions containing the function *Write*, where the optimization would also change the behaviour of the program.

6 Appendices

6.1 Task 1 - TypeChecker.sml

```
1 and checkExp ftab vtab (exp : In.Exp) = case exp of
2   ...
3   | In.And (e1, e2, pos)
4     => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Bool
5         , e1, e2)
6         in (Bool,
7             Out.And (e1_dec, e2_dec, pos))
8         end
9   | In.Or (e1, e2, pos)
10    => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Bool
11        , e1, e2)
12        in (Bool,
13            Out.Or (e1_dec, e2_dec, pos))
14        end
15  | In.Mult (e1, e2, pos)
16    => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int,
17        e1, e2)
18        in (Int,
19            Out.Mult (e1_dec, e2_dec, pos))
20        end
21  | In.Divide (e1, e2, pos)
22    => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int,
23        e1, e2)
24        in (Int,
25            Out.Divide (e1_dec, e2_dec, pos))
26        end
27  | In.Negate (e1, pos)
28    => let val (_, e1_dec) = checkUnOp ftab vtab (pos, Int, e1)
29        in (Int,
30            Out.Negate (e1_dec, pos))
31        end
32  | In.Not (e1, pos)
33    => let val (_, e1_dec) = checkUnOp ftab vtab (pos, Bool, e1)
34        in (Bool,
35            Out.Not (e1_dec, pos))
36        end
```

6.2 Task 1 - Interpreter.sml

```
1 fun evalExp ...
2   | evalExp ( Not(e, pos), vtab, ftab ) =
3       let val res    = evalExp(e, vtab, ftab)
4       in evalUnopBool(not, res, pos)
5       end
6   | evalExp ( And(e1, e2, pos), vtab, ftab ) =
```

```

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

```