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:

```
 fun keyword (s, pos) =
      case s of
...
          | "not"          => Parser.NOT pos
          | "true"         => Parser.TRUE pos
          | "false"        => Parser.FALSE pos
```

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

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

From Parser.grm:

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

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

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 endLabel = newName "andend"
            val code1 = compileExp e1 vtable place
            val code2 = compileExp e2 vtable place
        in code1 @ [ Mips.BEQ(place, "0", endLabel) ] @ code2 @ [Mips.LABEL endLabel]
        end
```

We based this on the following assembly code (for And):

```
    res_reg = checkexp1
    beq result,$0,end
    res_reg = checkexp2
    end
```

The idea is that if the first expression is false, the result will be false as well and if the first expression is true, the result will depend on (be equal to) the result of the second expression.

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

```
end
```

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

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

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

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, considering we haven't implementet assignment 5.
When running fasto with -i:

Mult: (mult.fo)
The function returns an int when getting two integers as input.
It gives an interpreter error on chars, strings and booleans.

Div: (div.fo)
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.

And: (And.fo, AndFail.fo)
Returns the correct boolean values depending on input, and returns an interpreter error on chars, strings.

Or: (Or.fo)
Returns the correct boolean values depending on input, and returns an interpreter error on chars, strings.

AndOr: (AndOr.fo)
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)
Returns the correct boolean values depending on input, and returns an interpreter error on chars, strings.

Negate: (neg_simple.fo)
Returns the correct boolean values.

True When running with -c:

Mult:
The function returns an int when getting two integers as input.
It gives an error when running in mars with any other input.

Div:
Returns the value of two integers divided. Returns an error on division by 0 and chars, and strings.

And:
Returns the true or false depending on the input values(0 or 1), on all other input it returns an error.

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

Or:
Returns the true or false depending on the input values(0 or 1), on all other input it returns an error.

AndOr:
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:
Returns the true on false and false on true and all other integers. It returns an error on chars and strings.

Negate:
Returns the correct boolean values.

3 Task 2 - Filter and Scan

The biggest challenge we encountered when implementing Filter and Scan was making sense of the variable names already used in CodeGen.sml. It took some time to find out which of the pointers (arr_reg, addr_reg, res_reg, elem_reg) pointed to what.

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 if the function call evaluated to true.

This was our strategy for the structure of the loop:

```
condition
beq condition ,$0 ,dontcopy
[copy code here]
dontcopy:
[increment here]
j condition
```

We decided on this structure as it has as few jumps as we think is possible.

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.

Of course we had to make some additions to the functionality of Map.

Here is our strategy for Scan:
First we add one to the size of the new array (as the output array will always be one element longer than the input),
then we evaluate our base element and copy it to the first slot in the new array as well as saving the value in a temporary register.
We then enter the loop where we call the passed function with our temporary base value and the first element in the input array as parameters, save the result to the temporary base value register and copy it to the new array.
And repeat until we run out of elements.
See Appendix Scan - CodeGen.sml for the full implementation.

3.3 Tests

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.

```
  | checkFunArg (In.Lambda (ret_type, params, exp, funpos) , vtab, ftab, pos) =
          let val Out.FunDec (fname, ret_type, args, body, pos) =
                  checkFunWithVtable (In.FunDec ("anon", ret_type, params, exp, fun
              val arg_types = map (fn (Param (_, ty)) => ty) args
              in (Out.Lambda (ret_type, args, body, funpos), ret_type, arg_types)
          end
```

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.

```
    | evalFunArg (Lambda (ret_type, args, body, funpos), vtab, ftab, callpos) =
          (fn aargs => callFunWithVtable (FunDec ("anon", ret_type, args, body, fun
```

In CodeGen.sml our extra case is, again, similar to that of normal function parameters, with the addition of the code to get the arguments and compile the body of the function (which in the other case is already done) and the extra labels needed for this.

```
    | applyFunArg (Lambda (ret_type, args, body, funpos), aargs, vtab, place, callpos
      let val tmp_reg = newName "tmp_reg"
          val funlabel = newName "anon_fun"
          val fun_res = newName "anon_fun_res"
          val (argcode, vtable_local) = getArgs args vtab minReg
          val compile_body = compileExp body vtable_local fun_res
      in  applyRegs(funlabel, aargs, tmp_reg, callpos)
        @ [Mips.LABEL funlabel]
        @ argcode
        @ compile_body
        @ [Mips.MOVE(place, fun_res)]
      end
```

4.1 Tests

5 Task 4 - ConstCopyPropFold

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.

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

In the case of Let we call copyConstPropFoldExp on the expression recursively and then add it to our vtable if it's foldable (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.

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

5.1 Shadowing

An example of shadowing:

```
     let x = 5 in
     let x = f(x) in
     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.

5.2 Tests

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
  | 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
  | 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
        end
  | evalExp ( Negate(e, pos), vtab, ftab ) =
        let val res   = evalExp(e, vtab, ftab)
        in  evalUnopNum(op ~, res, pos)
        end
  | evalExp ( Mult(e1, e2, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  evalBinopNum(op *, 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  evalBinopNum(op Int.quot, res1, res2, pos)
        end
```

6.3 Filter - CodeGen.sml

```
 | Filter (farg, arr_exp, elem_type, pos) =>

     let val size_reg = newName "size_reg" (* size of input array *)
         val arr_reg  = newName "arr_reg" (* address of array *)
         val elem_reg = newName "elem_reg" (* address of single element *)
         val res_reg = newName "res_reg" (* vrdi fra input arr og resultat af funktio
         val arr_code = compileExp arr_exp vtable arr_reg

         val get_size = [ Mips.LW (size_reg, arr_reg, "0") ]

         val addr_reg = newName "addr_reg" (* address of element in new array *)
         val i_reg = newName "i_reg"
         val new_counter = newName "zc"
         val init_regs = [ Mips.ADDI (addr_reg, place, "4")
                         , Mips.MOVE (i_reg, "0")
                         , Mips.ADDI (elem_reg, arr_reg, "4") ]

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

         val loop_map0 =
             let val crlabel = newName "cond_result"
                 val code0 = case getElemSize elem_type of
                                 One => [ Mips.LB(res_reg, elem_reg, "0") ]
                               | Four => [ Mips.LW(res_reg, elem_reg, "0") ]

                 val code1 = applyFunArg(farg, [res_reg], vtable, crlabel, pos)
                 val dontCopyLabel = newName "increment"

                 val copycode = case getElemSize elem_type of
                                 One => [ Mips.SB(res_reg, addr_reg, "0") ]
                                        @ [ Mips.ADDI(addr_reg, addr_reg, "1"), Mips.
                                | Four => [ Mips.SW(res_reg, addr_reg, "0") ]
                                         @ [ Mips.ADDI(addr_reg, addr_reg, "4") , Mip
                 val incrementcode = case getElemSize elem_type of
                                     One => Mips.ADDI (elem_reg, elem_reg, "1")
                                   | Four => Mips.ADDI (elem_reg, elem_reg, "4")
             in
                 code0 @ code1 @ [Mips.BEQ (crlabel,"0",dontCopyLabel)] @ copycode @
             end
         val write_new_size =
```

```
                [ Mips.SW (new_counter,place,"0") ]
            val loop_footer =
                [
                 Mips.ADDI (i_reg, i_reg, "1")
                , Mips.J loop_beg
                , Mips.LABEL loop_end
                ]
        in arr_code
           @ get_size
           @ dynalloc (size_reg, place, elem_type)
           @ init_regs
           @ loop_header
           @ loop_map0
           @ loop_footer
           @ write_new_size
        end
```

6.4 Scan - CodeGen.sml

```
  | Scan (farg, acc_exp, arr_exp, tp, pos) =>
        let val size_reg = newName "size_reg" (* size of input array *)
            val size_reg = newName "size_reg" (* size of output array *)
            val arr_reg  = newName "arr_reg" (* address of new array *)
            val elem_reg = newName "elem_reg" (* address of single element *)
            val res_reg = newName "res_reg" (* vrdi fra input arr og resultat af funktio
            val e_reg = newName "e_reg" (* vores udregnede vrdi der skal bruges i nste i
            val arr_code = compileExp arr_exp vtable arr_reg
            val acc_code = compileExp acc_exp vtable e_reg      (* her udregner vi vores

            val get_size = [ Mips.LW (size_reg, arr_reg, "0"),
                             Mips.ADDI(size_reg,size_reg,"1") ] (* fordi det nye array e

            val addr_reg = newName "addr_reg" (* address of element in new array *)
            val i_reg = newName "i_reg"
            val init_regs = let
                                val first_elem = case getElemSize tp of
                                    One =>  [ Mips.SB (e_reg, addr_reg, "0"),
                                              Mips.ADDI (addr_reg, addr_reg, "1") ]
                                  | Four => [ Mips.SW (e_reg, addr_reg, "0"),
                                              Mips.ADDI (addr_reg, addr_reg, "4") ]
                            in
                            [ Mips.ADDI (addr_reg, place, "4") ]
                            @ first_elem
                            @ [ Mips.MOVE (i_reg, "0")
                            , Mips.ADDI (elem_reg, arr_reg, "4") ]
                            end
            val loop_beg = newName "loop_beg"
            val loop_end = newName "loop_end"
            val tmp_reg = newName "tmp_reg"
            val loop_header = [ Mips.LABEL (loop_beg)
```

```
                                , Mips.SUB (tmp_reg, i_reg, size_reg)
                                , Mips.BGEZ (tmp_reg, loop_end) ]

            val loop_scan0 =
                case getElemSize tp of
                    One =>  Mips.LB   (tmp_reg, elem_reg, "0")
                            :: applyFunArg(farg, [e_reg, tmp_reg], vtable, res_reg, pos)
                            @ [ Mips.MOVE (e_reg, res_reg) ]
                            @ [ Mips.ADDI (elem_reg, elem_reg, "1") ]
                  | Four => Mips.LW   (tmp_reg, elem_reg, "0")
                            :: applyFunArg(farg, [e_reg, tmp_reg], vtable, res_reg, pos)
                            @ [ Mips.MOVE (e_reg, res_reg) ]
                            @ [ Mips.ADDI (elem_reg, elem_reg, "4") ]

            val loop_scan1 =
                case getElemSize tp of
                    One => [ Mips.SB (res_reg, addr_reg, "0") ]
                  | Four => [ Mips.SW (res_reg, addr_reg, "0") ]

            val loop_footer =
                [ Mips.ADDI (addr_reg, addr_reg,
                             makeConst (elemSizeToInt (getElemSize tp)))
                , Mips.ADDI (i_reg, i_reg, "1")
                , Mips.J loop_beg
                , Mips.LABEL loop_end
                ]
        in arr_code
           @ acc_code                     (* compile vores frste e-element *)
           @ get_size
           @ dynalloc (size_reg, place, tp)
           @ init_regs
           @ loop_header
           @ loop_scan0
           @ loop_scan1
           @ loop_footer
        end
```