# Oversættere Group Assignment

Martin Thiele, Alexander Mathiesen, Daniel Eyþórsson

20. december 2015

Studienummer: mqn507, pkq530, kzs773

Email: <Studienummer> @alumni.ku.dk

## 1 Introduction

We have decided to divide this report into three parts, one for each task. Each of these parts will then cover the different implementations we have made, the changes we have made to the different files, as well as any problems we may have encountered.

## 2 Warm up implementations

For the first task of the assignment we were asked to implement multiplication, division, boolean operators and literals. We have modified the following files: Lexer.lex, Parser.grm, Interpreter.sml, TypeChecker.sml and CodeGen.sml.

### 2.1 Boolean literals

#### 2.1.1 Lexer.lex

We added "true" and "false" by adding the tokens known as "TRUE" and "FALSE".

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

### 2.1.2 Parser.grm

We modified the token that handles many other keywords such as "if", "then", "else", etc. to also handle "true" and "false":

```
%token <(int*int)> IF THEN ELSE LET IN INT BOOL CHAR EOF AND OR NOT NEGATE TRUE FA
```

We now need to add these to their expression cases. Since they are boolean constants, they take no expressions, and this gives us to the following:

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

### 2.1.3 Interpreter.sml

No modifications were needed in this file.

### 2.1.4 TypeChecker.sml

No modifications were needed in this file.

### 2.1.5 CodeGen.sml

We implemented booleans in the "Constant" case of the "compileExp" function. It takes a boolean expression *b* and a position as parameters. If *b* evaluates to true, then it loads the value 1, and 0 otherwise. It uses the MIPS instruction "load immediate":

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

## 2.2 Multiplication and division

### 2.2.1 Lexer.lex

We added "*" and "/" by adding the tokens known as "MULT" and "DIV".

```
       | "*"            => Parser.MULT pos
       | "/"            => Parser.DIV pos
```

### 2.2.2 Parser.grm

We modified the token that handles the binary operators such as "plus" and "minus" etc. to also handle multiplication and division:

```
%token <(int*int)> PLUS MINUS DEQ EQ LTH MULT DIV
```

### 2.2.3 Interpreter.sml

We added cases to the "evalExp" function to handle multiplication ("Times") and division ("Divide"). They take as parameters two expressions, a position and two symbol tables. They then evaluate the two expressions and make sure that both are integer types before carrying out the respective operation:

```
  | 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
```

### 2.2.4 TypeChecker.sml

We added cases to the "checkExp" function to handle multiplication ("Times") and division ("Divide"). They take as parameters two expressions and a position. Then the "checkBinOp" function is invoked to make sure both expressions are of the same, correct type:

```
    | 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
```

### 2.2.5 CodeGen.sml

We implemented multiplication and division by adding them as cases to the "compileExp" function. It then evaluates each expression and invokes the Mips instructions "MUL" and "DIV" respectively.

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

### 2.2.6 Tests

## 2.3 AND and OR

### 2.3.1 Lexer.lex

We added "and" and "or" by adding the symbols "&&" and "||".

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

```
  | "||"                { Parser.OR     (getPos lexbuf) }
```

### 2.3.2 Parser.grm

We modified the token that handles many other keywords such as "if", "then", "else", etc. to also handle "and" and "or":

```
%token <(int*int)> IF THEN ELSE LET IN INT BOOL CHAR EOF AND OR NOT NEGATE TRUE FA
```

We then added associativity for both of them:

```
%left  OR AND
```

In order for the two expressions to work, they need to be implemented as expressions later on in the parser. We have done this by doing the following:

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

### 2.3.3 Interpreter.sml

We added cases to the "evalExp" function to handle AND ("And") as and OR ("Or"). They take as parameters two expressions, a position and two symbol tables. They then evaluate the two expressions and make sure that we are dealing with BoolVals, before carrying out the operation:

```
  | evalExp (And (e1, e2, pos), vtab, ftab) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  case (res1, res2) of
              (BoolVal n1, BoolVal n2) => if n1 then BoolVal (n1 = n2)
                                                  else BoolVal false
            | _ => invalidOperands "And on non-integral args: "
                   [(Bool, Bool)] res1 res2 pos
        end

  | evalExp (Or (e1, e2, pos), vtab, ftab) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  case (res1, res2) of
```

```
            (BoolVal n1, BoolVal n2) => if not n1 then BoolVal (n1 <> n2)
                                                  else BoolVal true
          | _ => invalidOperands "Or on non-integral args: "
                 [(Bool, Bool)] res1 res2 pos
    end
```

### 2.3.4 TypeChecker.sml

We added cases to the "checkExp" function to handle and ("And") and or ("Or"). They take as parameters two expressions and a position. Then the "evalExp" function is invoked on the first expression to ensure that it has the type boolean. This is done to maintain a short circuit structure, as if the first expression is not a boolean, there is no need to evaluate the second. We then proceed to type check the second expression in the same way.

```
| In.And (e1, e2, pos)
  => let val (t1, e1_dec) = checkExp ftab vtab e1
      in if t1 = Bool
        then let val (t2, e2_dec) = checkExp ftab vtab e2
         in if t2 = Bool then (Bool, Out.And(e1_dec, e2_dec, pos))
            else raise Error (("Wrong type: " ^ ppType t2), pos) end
        else raise Error (("Wrong type: " ^ ppType t1), pos)
      end

| In.Or (e1, e2, pos)
  => let val (t1, e1_dec) = checkExp ftab vtab e1
      in if t1 = Bool
        then let val (t2, e2_dec) = checkExp ftab vtab e2
          in if t2 = Bool then (Bool, Out.Or(e1_dec, e2_dec, pos))
          else raise Error (("Wrong type: " ^ ppType t2), pos) end
      else raise Error (("Wrong type: " ^ ppType t1), pos)
      end
```

### 2.3.5 CodeGen.sml

We implemented "AND" as well as "OR" expressions by adding them to the "compileExp" function.

For the "AND" case, we first create a "falseLabel" which marks a false evaluation. We then load the value 0 as we assume the expression returns false. We then use the "BEQ" instruction in Mips to see if $t1 = 0$ (using the $0 register in Mips), in which case we know the "AND" expression can only return false, and we jump to the "falseLabel". This is to maintain a short circuit structure. We then use the "BNE" instruction to see if $t1 \neq t2$. Since we know at this point that $t1$ is true, if this returns true, then $t2$ must be false, and we jump to the "falseLabel".

For the "OR" case, we first create a "trueLabel" which marks a true evaluation. We then load the value 1 as we assume the expression returns true. We then use use the "BNE" instruction in Mips to see if $t1 \neq 0$, in which case we know the "OR" expression can only return true, and we jump to the "trueLabel". We then use the sae instruction to see if $t1 \neq t2$. Since we know at this point that $t1$ is false, if this returns false, then $t2$ must be true, and we jump to the "trueLabel".

```
| 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 falseLabel = newName "false"
    in  code1 @ code2 @
        [ Mips.LI (place, "0")
        , Mips.BEQ (t1,"$0",falseLabel)
        , Mips.BNE (t1,t2,falseLabel)
        , Mips.LI (place,"1")
        , Mips.LABEL falseLabel ]
    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 trueLabel = newName "true"
    in  code1 @ code2 @
```

```
        [ Mips.LI (place, "1")
        , Mips.BNE (t1,"$0",trueLabel)
        , Mips.BNE (t1,t2,trueLabel)
        , Mips.LI (place,"0")
        , Mips.LABEL trueLabel ]
    end
```

### 2.3.6 Tests

## 2.4 Not and negation

### 2.4.1 Lexer.lex

We added "not" and "negate" by adding the symbols "!" and "∼".

### 2.4.2 Parser.grm

We modified the token that handles many other keywords such as "if", "then", "else", etc. to also handle "not" and "negate":

```
%token <(int*int)> IF THEN ELSE LET IN INT BOOL CHAR EOF AND OR NOT NEGATE TRUE FA
```

We then added associativity for both of them:

```
%nonassoc NEGATE NOT
```

In order for the two expressions to work, they need to be implemented as expressions later on in the parser. We have done this by doing the following:

```
        | NOT         Exp  { Not    ($2, $1)     }
        | NEGATE      Exp  { Negate ($2, $1)     }
```

### 2.4.3 Interpreter.sml

We added cases to the "evalExp" function to handle not ("Not") as well as negate ("Negate"). They take as parameters an expression, a position and two symbol tables. They then evaluate the expression and make sure that we are dealing with a boolean or integer respectively, before carrying out the operation:

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

| evalExp ( Negate(e, pos), vtab, ftab ) =
      let val res = evalExp(e, vtab, ftab)
      in case res of IntVal n => IntVal(0-n)
          | _ => invalidOperands "Negate on non-integral arg: " [(Int, Int)] res
      end
```

### 2.4.4 TypeChecker.sml

We added cases to the "checkExp" function to handle not ("Not") and negate ("Negate"). They take as parameters an expression and a position. Then the "checkExp" function is invoked on the expression to ensure that it has the type boolean or integer respectively.

```
| In.Not (e, pos)
  => let val (t, e_dec) = checkExp ftab vtab e
      in if t = Bool then (Bool, Out.Not(e_dec, pos))
      else raise Error (("Wrong type: " ^ ppType t), pos)
      end

| In.Negate (e, pos)
  => let val (t, e_dec) = checkExp ftab vtab e
      in if t = Int then (Int, Out.Negate(e_dec, pos))
      else raise Error (("Wrong type: " ^ ppType t), pos)
      end
```

### 2.4.5 CodeGen.sml

We implemented "NOT" as well as "NEGATE" evaluations by adding them to the "compileExp" function.

For the "NOT" case we use the Mips instruction "XORI" on the boolean expression $t$ and the value 1. Because "XORI" will return true if, and only if, exactly one of the two expressions is true, and false otherwise, invoking it with one fixed, true expression means that if $t$ is false, "XORI" returns true, vice versa.

For the "NEGATE" case we use the Mips instruction "SUB" to subtract the integer $t$ from the value 0.

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

| Negate (e', pos) =>
    let val t = newName "not_arg"
        val code = compileExp e' vtable t
    in code @ [Mips.SUB (place, "$0", t)]
    end
```

### 2.4.6 Tests

## 2.5 MAP and REDUCE

### 2.5.1 Lexer.lex

We added "map" and "reduce" by adding the tokens known as "MAP" and "REDUCE".

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

### 2.5.2 Parser.grm

We added a token that handles "map" and "reduce":

```
%token <(int*int)> MAP REDUCE
```

In order for the two expressions to work, they need to be implemented as expressions later on in the parser. We have done this by doing the following:

```
| MAP LPAR FunArg COMMA Exp RPAR
                { Map ($3, $5, (), (),  $1) }
```

```
        | REDUCE LPAR FunArg COMMA Exp COMMA Exp RPAR
                        { Reduce ($3, $5, $7, (), $1)}
```

### 2.5.3 Interpreter.sml

We added cases to the "evalExp" function to handle map ("Map") as well as reduce ("Reduce"). Map takes as parameters a function argument, an array expression position and two symbol tables. Reduce takes as parameters a function argument, an initial value, an array expression, a type expression, a position and two symbol tables. They then evaluate the evaluate the function and array expressions and make sure the types match.

```
  | evalExp ( Map (farg, arrexp, _, _, pos), vtab, ftab ) =
        let val arr  = evalExp(arrexp, vtab, ftab)
            val farg_ret_type = rtpFunArg (farg, ftab, pos)
        in case arr of
               ArrayVal (lst,tp1) =>
               let val mlst = map (fn x => evalFunArg (farg, vtab, ftab, pos, [x])
               in  ArrayVal (mlst, farg_ret_type)
               end
             | _ => raise Error("Map: Wrong argument: " ^ppVal 0 arr, pos)
        end

  | evalExp ( Reduce (farg, ne, arrexp, tp, pos), vtab, ftab ) =
            let val arr  = evalExp(arrexp, vtab, ftab)
                val e  = evalExp(ne, vtab, ftab)
                val farg_ret_type = rtpFunArg (farg, ftab, pos)
        in case arr of
               ArrayVal (lst,tp1) =>
               foldl (fn (x,y) => evalFunArg (farg, vtab, ftab, pos, [x,y])) e lst
             | _ => raise Error("Reduce: Wrong argument: " ^ppVal 0 arr, pos)
        end
```

### 2.5.4 TypeChecker.sml

We added cases to the "evalExp" function to handle map ("Map") as well as reduce ("Reduce"). Map takes as parameters a function argument, an array expression position and two symbol tables. Reduce takes as parameters a function argument, an initial value, an array expression, a type expression, a position and two symbol tables.

We then evaluate the expressions of both function and array expressions to make sure they match. Anonymous functions are accounted for by use of a helper function called "checkFunArg", which handles both Lambda and standard functions.

```
| In.Map (f, arr_exp, _, _, pos)
  => let val (a_type, arr_exp_dec) = checkExp ftab vtab arr_exp
         val e_type =
          case a_type of Array r => r
                      |  _ => raise Error("Map: wrong argument type " ^ ppType
         val (f', f_ret, f_arg) =
          case checkFunArg (f, vtab, ftab, pos) of
            (f', ret, [t]) => (f', ret, t)
           | (_,  ret, args) => raise Error("Map: wrong argument type " ^ ppTy
       in if e_type = f_arg
          then (Array f_ret, Out.Map (f', arr_exp_dec, e_type, f_ret, pos))
          else raise Error ("Map: incompatible arguments " ^ ppType e_type, pos
       end

| In.Reduce (f, n_exp, arr_exp, _, pos)
  => let val (a_type, arr_exp_dec) = checkExp ftab vtab arr_exp
         val e_type =
          case a_type of Array r => r
                      |  _ => raise Error("Reduce: wrong argument type " ^ ppT
         val (n_type, n_exp_dec) = checkExp ftab vtab n_exp
         val (f', f_ret, f_arg) =
          case checkFunArg (f, vtab, ftab, pos) of
            (f', ret, [t]) => (f', ret, t)
           | (_,  ret, args) => raise Error("Reduce: wrong argument type " ^ p
       in if (e_type = f_arg andalso n_type = f_arg)
          then (e_type, Out.Reduce (f', n_exp_dec, arr_exp_dec, e_type, pos))
          else raise Error ("Reduce: incompatible arguments " ^ ppType e_type,
       end
```

### 2.5.5 CodeGen.sml

We implemented "MAP" as well as "REDUCE" evaluations by adding them to the "compileExp" function.

We initialize labels to help control the flow of the function. We iterate over the size of the array, and as the loop header we have used "BGEZ" instruction in Mips to check for remaining, previously unencountered elements in the array before each iteration.

Then in the loop body we apply the function to the element we are currently evaluating and storing the result in a register labeled "res_reg". We account for different datatypes and their difference in size by use of the function "getElemSize", and a case struct which uses "LB" or "LW" instructions in Mips depending on the data size we are working on.

```
| Map (farg, arr_exp, elem_type, ret_type, pos) =>
  let val size_reg = newName "size_reg"
      val arr_reg  = newName "arr_reg"
      val elem_reg = newName "elem_reg"
      val addr_reg = newName "addr_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 res_reg  = newName "res_reg"
      val arr_code = compileExp arr_exp vtable arr_reg

      val get_size = [ Mips.LW (size_reg, arr_reg, "0")]
      val init_regs = [ Mips.ADDI (addr_reg, place, "4")
                      , Mips.MOVE (i_reg, "0")
                      , Mips.ADDI (elem_reg, arr_reg, "4") ]
      val loop_header = [ Mips.LABEL (loop_beg)
                        , Mips.SUB (tmp_reg, i_reg, size_reg)
                        , Mips.BGEZ (tmp_reg, loop_end) ]

      val loop_map0 = case getElemSize elem_type of
  One  => Mips.LB(res_reg, elem_reg, "0")::
       applyFunArg(farg, [res_reg], vtable, res_reg, pos)
```

```
        @ [ Mips.ADDI(elem_reg, elem_reg, "1")]
| Four => Mips.LW(res_reg, elem_reg, "0")::
        applyFunArg(farg, [res_reg], vtable, res_reg, pos)
        @ [ Mips.ADDI(elem_reg, elem_reg, "4")]

    val loop_map1 = case getElemSize elem_type of
   One  => [ Mips.SB(res_reg, elem_reg, "0") ]
|  Four => [ Mips.SW(res_reg, elem_reg, "0") ]

      val loop_footer = [ Mips.ADDI (addr_reg, addr_reg, "4")
                        , Mips.ADDI (i_reg, i_reg, "1")
                        , Mips.J (loop_beg)
                        , Mips.LABEL (loop_end)]
  in arr_code
  @ get_size
    @ dynalloc(size_reg, place, ret_type)
    @ init_regs
    @ loop_header
    @ loop_map0
  @ loop_map1
    @ loop_footer
    end


(* reduce(f, acc, {x1, x2, ...}) = f(..., f(x2, f(x1, acc))) *)
| Reduce (binop, acc_exp, arr_exp, tp, pos) =>
  let val size_reg = newName "size_reg"
      val arr_reg  = newName "arr_reg"
      val addr_reg = newName "addr_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 arr_code = compileExp arr_exp vtable arr_reg
      val acc_code = compileExp acc_exp vtable arr_reg


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

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


        val loop_header = [ Mips.LABEL (loop_beg)
                          , Mips.SUB   (tmp_reg, i_reg, size_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.LB(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 (addr_reg, addr_reg, "4")
                          , Mips.ADDI (i_reg, i_reg, "1")
                          , Mips.J (loop_beg)
                          , Mips.LABEL (loop_end)]

    in arr_code
      @ get_size
      @ acc_code
      @ dynalloc(size_reg, place, Int)
      @ init_regs
      @ loop_header
      @ loop_reduce
      @ loop_footer
      end

(* Lambda helper function *)

and applyFunArg (FunName s, args, vtable, place, pos) : Mips.Prog =
      let val tmp_reg = newName "tmp_reg"
      in  applyRegs(s, args, tmp_reg, pos) @ [Mips.MOVE(place, tmp_reg)] end
```

```
| applyFunArg (Lambda (_, params, body, fpos), args, vtable, place, pos) =
  let val tmp_reg = newName "tmp_reg"
      fun bindArgToVtable (Param(pn, pt), arg, vtable) = SymTab.bind pn arg vt
      val vtable' = ListPair.foldr bindArgToVtable vtable (params, args)
      val code = compileExp body vtable' tmp_reg
  in
     code @ [Mips.MOVE(place, tmp_reg)]
  end
```

#### 2.5.6 Tests

### 2.6 Binary operators

Unfortunately we did not get around to this, and all we managed to do was implement the unknown BinOp in the parser.

## 3 Task 3