# 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 this 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. These both take a position as their only argument.

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

### 2.1.2 Parser.grm

We added a token "BOOLVAL" to the list and defined it to take two arguments; a boolean value and a position, which is a tuple of integers:

```
%token <(int*int)> TRUE FALSE
```

We now need to add these to their expression cases. Since they are booleans and constants, they don't take any expressions, and results 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 "/" cases to the keyword function. We assume the implementation in the parser phase takes a single argument; its position which is a tuple of integers:

```
        | "*"            => 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
```

## 2.3 AND and OR

### 2.3.1 Lexer.lex

We added "and" as well as "or" as symbols instead of text as "and" as well as "or" usually is being used with a different meaning when writing SML.

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

### 2.3.2 Parser.grm

We modified the token that handles if-then-else statements to also handle AND as well as OR

```
%token <(int*int)> AND OR
```

We followed the plus and minus case and determined that we needed to implement associatives for the tokens. thus:

```
%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 well as 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

In the typechecker we check if e1 is a boolean, this is done to short-circuit. If e1 is a boolean, then e2 is checked in the case of being another boolean. If it is then we continue on, else we raise an error

```
| 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 evaluations by adding them to the "compileExp" function. It then evaluates each expression and invokes the following mips instructions:

**AND**

```
[ Mips.LI (place, "0")          (*Assume t1 and t2 is unequal*)
, Mips.BEQ (t1,"$0",falseLabel) (*Compare t1 to 0*)
, Mips.BNE (t1,t2,falseLabel)   (*Compare t1 to t2*)
, Mips.LI (place,"1")           (*If not false at this point, true*)
, Mips.LABEL falseLabel ]       (*false label*)
```

**OR**

```
[ Mips.LI (place, "1")          (*Assume t1 and t2 is equal*)
, Mips.BNE (t1,"$0",trueLabel) (*Compare t1 to 0 for short-circuiting*)
, Mips.BNE (t1,t2,trueLabel)   (*Compare t1 to t2*)
, Mips.LI (place,"0")          (*If not true at this point, then false*
, Mips.LABEL trueLabel ]       (*True label*)
```

**Full code:**

```
| 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
```

## 2.4 Not and negation

### 2.4.1 Lexer.lex

We added not and negate to our lexer file by using the symbols "!" and "∼" respectively. There was no mention of this in the assignment, so we felt this would be most logical, seeing how these are already the recognized as their respective symbols in a lot of programming languages.

### 2.4.2 Parser.grm

We modified the token that handles if-then-else statements to also handle NOT as well as NEGATE

```
%token <(int*int)> NOT NEGATE
```

By using the same logic for plus and minus we had to add a precedence to our NOT and NEGATE cases. These are done as so

```
%nonassoc NEGATE NOT
```

And just as well, we need to add them in their expression cases as well. These only take one expression which differs from the previous ones

```
        | 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 BoolVal in **Not**, and an IntVal in **Negate**, 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
Dr        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

For the first case, not, we check if the expression is a boolean, if it is carry on, else raise error. In the second case of negate, we check if the expression is an integer, if it is we carry on.

```
| 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. It then evaluates each expression and invokes the following mips instructions:

**NOT**

```
[Mips.XORI (place,t, "1")]
```

**OR**

```
[Mips.SUB (place, "$0", t)]
```

**Full code:**

```
| 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
```

## 3 Array combinators

Not yet implemented