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" cases to the keyword function. We assume the implementation of boolean values in the parser phase to handle both "true" and "false" and take these as extra an extra parameter alongside the position:

```
| "true"  => Parser.BOOLVAL(true, pos)
| "false" => Parser.BOOLVAL(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 <(bool*(int*int))> BOOLVAL
```

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
```