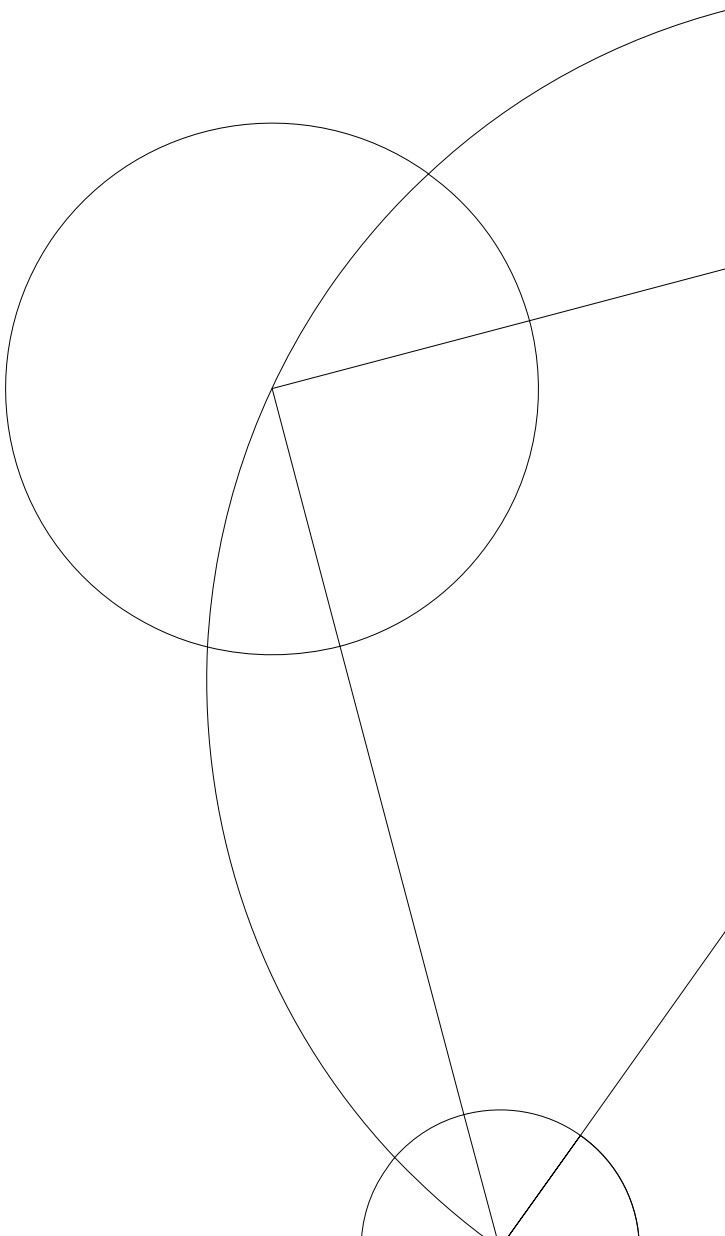
UNIVERSITY OF COPENHAGEN

Compiler Design

Compiler for the Fasto Programming Language

Magnus Nørskov Stavngaard, Mark Jan Jacobi and Christian Salbæk
magnus@stavngaard.dk, mark@jacobi.pm and chr.salbaek@gmail.com

December 9, 2014



1 Task 1 - Warmup

In task 1 we were asked to implement the boolean operators `&&`, `||` and `not`, the boolean constants true and false, integer multiplication, integer division and integer negation.

We will go through in detail the implementation of integer division and multiplication and then skip rather quickly over the implementation of the rest of the operators only describing what is different from multiplication and division as the operations is implemented very similar.

1.1 Integer Multiplication and Division

We started by implementing integer multiplication and division in the Lexer. We created a new rule for the star and division operator that created tokens and passed the tokens to the parser.

```
| '*'   { Parser.TIMES    (getPos lexbuf) }
| '/'   { Parser.DIVIDE   (getPos lexbuf) }
```

In the parser we added the tokens where addition and subtraction was already defined, as integer multiplication and division has allot in common with addition and subtraction. Integer multiplication and division carries two integers corresponding to a position in the code.

```
%token <(int*int)> PLUS MINUS TIMES DIVIDE DEQ EQ LTH BOOLAND BOOLOR NOT NEG
```

We also declare both times and divide as left associative operators with greater precedence than addition and subtraction.

```
%left BOOLOR
%left BOOLAND
%left NOT
%left DEQ LTH
%left PLUS MINUS
%left TIMES
%left DIVIDE
%left NEG
```

We then defined that an expression could consist of an expression followed by a multiplication or division followed by an expression. And that this correspond to `Times` and `Divide` in the Fasto language definition.

```
Exp :     NUM                 { Constant (IntVal (#1 $1), #2 $1) }
        | CHARLIT             { Constant (CharVal (#1 $1), #2 $1) }

        (...)

        | Exp TIMES Exp       { Times($1, $3, $2) }
        | Exp DIVIDE Exp      { Divide($1, $3, $2) }

        (...)
```

In the interpreter we implemented cases for `Times` and `Divide` in the `evalExpr` function.

```
| evalExp ( Times(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
```

Our cases evaluate recursively the expressions to the left and right of the operator and then calls `evalBinopNum` with the appropriate operator and the results from evaluating the left-hand and the righthand side of the expression.

We then implemented the operators in the typechecker. Our cases call a helper function `checkBinOp` that takes a position, an expected type, and two expressions and check that the two expressions have the type of the expected type. If the types match the types is returned with *typedecorated* versions of the expressions, if the types doesn't match an error is raised.
We then simply return the same operation, now with a return 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
```

We can now finally implement the operators in the code generator. Here we create two temporary variables `t1` and `t2` to hold the values of the expression on either side of the operator. We then call the function `compileExp` recursively with these names to get the machine code for the expression on either side of the operator. Then we just simply return a list of first the code to compute the left hand side of the operator, then the right hand side, and then we apply the MIPS commands `MUL` and `DIV` to the two *subresults* and save the result in `place`.

```
| Times (e1, e2, pos) =>
    let val t1 = newName "times_L"
        val t2 = newName "times_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 "divide_L"
        val t2 = newName "divide_R"
        val code1 = compileExp e1 vtable t1
        val code2 = compileExp e2 vtable t2
    in  code1 @ code2 @ [Mips.DIV (place, t1, t2)]
    end
```