UNIVERSITY OF COPENHAGEN

Compiler Design

Compiler for the Fasto Programming Language

Magnus Nørskov Stavngaard
magnus@stavngaard.dk

Mark Jan Jacobi
mark@jacobi.pm

Christian Salbæk
chr.salbaek@gmail.com

December 22, 2014

Contents

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 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
```

1.2 Boolean Operators

We started by implementing the boolean operators in the lexer in a very similar way that we implemented multiplication and division. In the parser, we made sure that the boolean operators were defined as having a lower precedence than the arithmetic operators, so that an expression like,

```
        2 + 4 == 6 || 5 + 8 == 10 && 8 < 10
```

is evaluated like,

```
        ((2 + 4) == 6) || (((5 + 8) == 10) && (8 < 10)).
```

Notice that the `&&` operator is also evaluated before the `||` operator.

After this we implemented the operators in the interpreter. Here we created a case for `and` and a case for `or`. We had to implement them as short circuited which means that the right hand side of an `and` should only be evaluated if the left hand side is true. Similarly for `or` the right hand side should only be evaluated if the left hand side is false (The or implementation can be seen in appendix A.

```
| evalExp ( And(e1, e2, pos), vtab, ftab ) =
        let val r1 = evalExp(e1, vtab, ftab)
        in case r1 of
           BoolVal b1 => if b1 then
                           let val r2 = evalExp(e2, vtab, ftab)
                           in case r2 of
                              BoolVal b2 => BoolVal b2
                            | otherwise  => raise Error ("And expect boolval", pos)
                           end
                         else BoolVal b1
         | otherwise  => raise Error ("And expect boolval", pos)
        end
```

We first evaluate the lefthand expression, if that is true we evaluate the right hand expression, if that is also true we return True otherwise we return false. If either of the expressions isn't a BoolVal, we report an error.

In the code generator it was also required that we implemented the boolean operators to be short-circuiting so that the right hand side of an `and` only evaluates if the left hand side is true. Similarly the right hand side of an `or` evaluates only if the left hand side is false. We did this with the MIPS assembly code,

```
$t1 = compile e1

beq $t1, $zero, False

$t2 = compile e2

beq $t2, $zero, False
li  $s1, 1 ;; Assuming the result should be saved to $s1
j   End

False:
    li  $s1, 0 ;; Assuming the result should be saved to $s2

End:
```

It can be seen that if the first part of an `and` return false i.e. the result register contains 0, we simply skip over the execution of the right hand side and jump strait to False. We did something similar for `or`'s. The Standard ML code generating the MIPS assembly for both `and` and `or` can be seen in appendix B.

1.3 Boolean Negation

In the lexer we implemented the operator for boolean negation `not` as a keyword. We did this because `not` is a valid variable name and would pass the rule,

```
| [`a`-`z` `A`-`Z`] [`a`-`z` `A`-`Z` `0`-`9` `_`]*
                                  { keyword (getLexeme lexbuf,getPos lexbuf) }.
```

If we didn't implement a keyword, `not` would simply fall through and go in the case,

```
| _                   => Parser.ID (s, pos)
```

in the `keyword` function. The implemented keyword goes to a `Parser.NOT` in the parser and carries only the position.

```
fun keyword (s, pos) =
    case s of
        "if"            => Parser.IF pos
      | "then"          => Parser.THEN pos

      (...)

      | "not"           => Parser.NOT pos
      | "fn"            => Parser.LAMBDA pos
      | _               => Parser.ID (s, pos)
}
```

In the interpreter `not` is implemented simply by evaluating the expression after the `not`. If that expression results in a true, false is returned, if it results in a false, true is returned and otherwise an error is reported. The code can be seen in appendix C.

`not` is implemented in the code generator as a branch operation. If `not` is applied to true 3 operations are performed, if it is applied to false only 2 operations are performed. The code generating MIPS assembly can be found in appendix D.

1.4 Integer Negation

Integer negation has been implemented two places in the lexer. For integer constants which is negated the negated value is simply created on compiletime. This is done by the rule,

```
| [`0`-`9`]+ | "~" [`0`-`9`]+ { case Int.fromString (getLexeme lexbuf) of
                                  NONE    => lexerError lexbuf "Bad integer"
                                | SOME i => Parser.NUM (i, getPos lexbuf) }
```

which says that a number or a tilde followed by a one or more numbers is a Parser.NUM in the parser and the integer value is carried with it.

If a tilde is not followed by a number for example in,

```
~(2 + 4) - ~(2 - ~f(3))
```

the negation is caught by the rule,

```
| "~"                   { Parser.NEG        (getPos lexbuf) }
```

In the code generator the negations is performed as a exclusive or and an addition.

```
| Negate (e, pos) =>
    let val negThis  = newName "negThis"
        val code     = compileExp e vtable negThis
        val negation =
            [Mips.XORI(negThis,negThis,"-1")] @ [Mips.ADDI (place,negThis,"1")]
    in code @ negation
    end
```

This works because computers use two's complement to express numbers. In two's complement the negation of a number is computed by flipping all bits in the number and adding 1. We flip all bits by exclusive or'ring with -1, the binary value of -1 is 1111 1111. Therefore all the places where there was 0 in the original number there will now be 1 and where there were 1 there will now be 0. After that we simply add 1 with `addi` and save the result to place.

1.5 Boolean Literals

The boolean literals is implemented in the lexer as a keyword that carries a boolean value to the parser. The implementation of boolean literals in all compiler phases can be found in appendix E.

1.6 Test

Test name	Test description
andOr.fo	Tests boolean operators and their precedence.
and_sc.fo	Test if and is properly short circuited.
boolCompare.fo	Test boolean literals.
boolLit.fo	Another test testing if boolean literals.
intNegate.fo	Test negation of integer values, only test the simple case number not (number). Operator precedence is also tested.
muldivide.fo	Tests the multiplication and division operators.
negate2.fo	Again test simple integer negation.
negate.fo	Test the case where integer negation is followed by something other than a number.
or_sc.fo	Test if or is short circuited.
tobeornottobe.fo	Test precedence of boolean operators.

2 Task 2 - Implement `filter` and `scan`

2.1 Typerules

The typerules for `filter` and `scan` have been based on the already existing typerules for `map` and `reduce`. They are as follows:

`filter`: $(\alpha \rightarrow \text{bool}) * [\alpha] \rightarrow [\alpha]$, typerule for `filter(f, x)`:

- compute t, the type of x and check that $t = [t_e]$ for some type t_e
- get f's signature from `ftable`. `IF` f does not receive exactly one argument `THEN` return `error() ELSE` f: $t_{in} \rightarrow t_{out}$ for some types t_{in} and t_{out}.
- `IF` $t_{in} = t_e$ `AND` $t_{out} =$ `bool THEN filter(f, x) ELSE error()`.

`scan`: $(\alpha * \alpha \rightarrow \alpha) * \alpha * [\alpha] \rightarrow [\alpha]$, typerule for `scan(f, e, x)`:

- Compute t, the type of e and t_x, the type of x and check that:
    1. f: $(t * t) \rightarrow t$
    2. $t_x = [t]$
- If so then `scan(f, e, x)`

3 Task 3 - λ-expressions in SOAC's

3.1 Lexer

We implemented lambda functions in the lexer by creating a keyword `fn` corresponding to a `LAMBDA` in the parser. We also created a token for the special equals symbols used in lambda expressions (=>). We called this token `LAMBDAEQ`.

3.2 Parser

We then implemented lambdas in the parser. We did this by observing that the `map` function in the parser is defined as,

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

meaning that the lambda is supposed to passed as a `FunArg`. We then looked at the declaration of a FunArg,

```
FunArg : ID { FunName (#1 $1) },
```

and added to this definition a case for a lambda function. In Fasto a `FunArg` is defined as,

```
and FunArg = Lambda of Type * Param list * Exp * pos
           | FunName of string.
```

We then just took the syntax of a lambda and translated that to a list of tokens. Then we pattern matched on that expression and transfered the values needed by the lambda in Fasto. Furthermore, Lambda expressions can also be called without arguments, therefore we need a case where the Params is non existant.

Listing 1: Lambda in Parser.grm

```
FunArg : ID { FunName (#1 $1) }
       |  LAMBDA Type LPAR Params RPAR LAMBDAEQ Exp { Lambda ($2, $4, $7, $1) }
       |  LAMBDA Type LPAR RPAR LAMBDAEQ Exp { Lambda ($2, [], $6, $1) }
```

3.3 Interperter

We implemented lambdas in the interpreter by changing the way function arguments are evaluated. We could do this as it is an invariant of the Fasto programming language that lambdas can only be used in Second Order Array Constructors (SOAC's). We simply matched a case where `evalFunArg` is called with a lambda instead of a function name. We then use this lambda to construct a function definition with the generic name *lambda*. We then call `callFunWithVtable` with this function declaration and the vtable passed to the function, this means that we keep the binding between local variable names and their values and the lambda can use these variables. The function returns an anonymous function in sml that takes an argument list and applies the lambda to those arguments.

```
and evalFunArg (FunName fid, vtab, ftab, callpos) =
    let
      val fexp = SymTab.lookup fid ftab
    in
      case fexp of
        NONE   => raise Error("Function "^fid^" is not in SymTab!", callpos)
      | SOME f => (fn aargs => callFun(f, aargs, ftab, callpos), getFunRTP f)
    end
  | evalFunArg (Lambda (tp, paralist, exp, pos), vtab, ftab, pcall) =
    let val fexp = FunDec ("lambda", tp, paralist, exp, pos)
    in (fn aargs => callFunWithVtable(fexp, aargs, vtab, ftab, pcall), tp)
    end
```

3.4 Code generator

The code generation of lambda is a bit complex. The easiest way to explain how it works is by a list, where each step is followed by the next, in chronological order.

1. Remove type decleration from paralist.

2. Zip the newly generated list with args.
3. Generate code that assings input from args to the parameter list.
4. Generate the local vTable, that contains the local variables.
5. Compile the expression from the lambda, via compileExp. It takes the local vTable, and a register to place its result.
6. then we concatinate the code from 3 and 5, and returns this.

The source code looks like this

Listing 2: Lambda in CodeGen.sml

```
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 (tp, paralist, exp, _), args, vtable, place, pos) =
    let val lambda = newName "lambda"
        val argsparalist = map (fn Param(x,y) => x) paralist (*Removes type
            decleration*)
        val zipped = zipWith (fn (x,y)=>(x,y)) argsparalist args (*Zips the
            arguments*)
        val bind = zipWith Mips.MOVE argsparalist args (*Moves the arguments
            *)
        (*Delcares the local vtable, by binding zipped in a SymTab*)
        val localVtable = foldl(fn ((x,y),acc)=>SymTab.bind x y acc) (SymTab
            .empty()) zipped
        (*compiles the lambda expression with the local v table, and saves
            res to place*)
        val compiledExp = compileExp exp localVtable place
    in []
      @ bind
      @ compiledExp
    end
```

4 Task 4 - Copy propagation and constant folding

For our implementation of the optimizations copy propagration and constant folding, we have made additions to `CopyConstPropFold.sml`, specifically we have added cases for variables, `let`-bindings, multplication, division, logical and/or, logical negation and integer negation in function `copyConstPropFoldExp`.

4.1 Multiplication

Our implementation of the case for multiplication is based on the simplifications given for constant folding of multiplication expressions given in `GroupProj14.pdf`. They are as follows, here `e1'` and `e2'` are the recursively optimized subexpressions, found in a multiplication expression: `e1 * e2`

- if `e1'` and `e2'` are constant values then the result will be the multiplication value,
- if `e1'` (`e2'`) is value `1` then the result is `e2'` (`e1'`) because `1 * e2' = e1'`,
- if `e1'` (`e2'`) is value `0` then the result is constant value `0` because `0 * e2' = 0`,
- otherwise the optimsed result is `Times (e1', e2', pos)`.

Based on this we have implemented our optimization case for multiplication expressions like this:

```
| Times (e1, e2, pos) =>
  let val e1' = copyConstPropFoldExp vtable e1
      val e2' = copyConstPropFoldExp vtable e2
  in case (e1', e2') of
         (Constant (IntVal x, _), Constant (IntVal y, _)) =>
         Constant (IntVal (x*y), pos)
       | (Constant (IntVal 0, _), _) =>
         Constant (IntVal 0, pos)
       | (_, Constant (IntVal 0, _)) =>
         Constant (IntVal 0, pos)
       | (Constant (IntVal 1, _), _) =>
         e2'
       | (_, Constant (IntVal 1, _)) =>
         e1'
       | _ =>
         Times (e1', e2', pos)
  end
```

First the subexpressions, `e1` and `e2`, of the multiplication expression are optimized, and put respectively in variables `e1'` and `e2'`. Then the cases described above are run through. A similar rationale have been applied to the implementation of the case for division. The implementation can be seen in appendix **??**.

4.2 Logical and

For the case of logical and, we have made the following simplificatons, and based our implementation on these. Again `e1'` and `e2'` are the recursively optimized subexpressions:

- if `e1'` and `e2'` are bools, then the result should be `e1' and e2'`
- otherwise the optimsed result is `And (e1', e2', pos)`.

Based on this, we have implemented the case for logical and, in the following way:

```
| And (e1, e2, pos) =>
  let val e1' = copyConstPropFoldExp vtable e1
      val e2' = copyConstPropFoldExp vtable e2
  in case (e1', e2') of
         (Constant(BoolVal x,_),Constant(BoolVal y,_)) =>
         Constant(BoolVal (x andalso y),pos)
       | _ =>
         And (e1', e2', pos)
  end
```

First the subexpressions `e1` and `e2`, found in the and-expression, are optimized, and put respectively in variables `e1'` and `e2'`. The cases descriped above are then run through. Our implementation of logical or follows a similar rationale, and can be found in appendix **??**.

4.3 Logical not

As with multiplication and logical and, we have made some simplifications describing the constant folding on logical not, they are as follows:

- if `e1'` is true then return false,
- if `e1'` is false then return true,
- otherwise the optimsed result is `Not (e1', pos)`.

Our implementation has been based on these simplification, it is found below:

```
| Not (e1, pos) =>
  let val e1' = copyConstPropFoldExp vtable e1
  in case (e1') of
           Constant (BoolVal true, _) =>
           Constant (BoolVal false, pos)
         | Constant (BoolVal false, _) =>
           Constant (BoolVal true, pos)
         | _ =>
           Not (e1', pos)
  end
```

First `e1` is recursively optimised, and put in `e1'`. The described cases are then run through. Our implementation of integer negation follows a similar rationale, and can be found in appenddix **??**.

4.4 Variables

For variables we have implemented the following:

```
| Var (name, pos) =>
  let val name = name
      val pos = pos
  in case (SymTab.lookup name vtable) of
       SOME (VarProp newname) => Var (newname, pos)
     | SOME (ConstProp value) => Constant (value, pos)
     | _                      => Var (name, pos)
  end
```

First the given variable is looked up in the `vtable`. If a new variable name is returned, our variable is thus just a copy of that variable, and we return the new variable, i.e copy propagation. If on the other hand a constant value is returned, we return that constant, i.e constant propagation. Otherwise our variable is returned as it is, i.e no propagation occurs.

4.5 `let`-bindings

We use the term propagatee throughout this part. A propagatee is a variable's defining expression. The defining expression of a variable is defined to be either a constant value, or another variable. Below is our code for `let`-bindings. Generally this case detects propagatees in the `let`-binding expression and binds them to the vtable. It also optimizes the expression and the body of the `let`-binding.

```
| Let (Dec (name, e, decpos), body, pos) =>
  let val e' = copyConstPropFoldExp vtable e
      val vtable' = bindExpPropagatee name e' vtable
      val body' = copyConstPropFoldExp vtable' body
  in Let (Dec (name, e', decpos), body', pos)
  end
```

Propagatees are detected and bound to the vtable, by calling function `bindExpPropagatee` on the optimized expression `e'` and the `vtable`. `bindExpPropagatee` calls another function `expPropagatee`, which extracts an propagatee from a given expression. If some propagatee is returned from `expPropagatee`, it is bound to the vtable.

```
fun expPropagatee (Var (varname, _)) = SOME (VarProp varname)
  | expPropagatee (Constant (value, _)) = SOME (ConstProp value)
  | expPropagatee _ = NONE

fun bindExpPropagatee name e vtable =
    case expPropagatee e of
        NONE => SymTab.remove name vtable
      | SOME prop => SymTab.bind name prop vtable
```

4.6 Tests

We have tested out optimization implementations by using the commands:
`\bin\fasto -p c testfile.fo` and `\bin\fasto -p testfile.fo`. With `-p c`, we only test with the constant folding and copy/constant propagation pass, whereas with just `-p`, we run the only set of optimizations. Our first example is found in `1optim.fo`, here we first test with `-p c`.

```
fun int main() =                          fun int main() =
    let a = read(int) in                      let a = read(int) in
    let b =                                   let b =
        let x = a in             →                let x = a in
        let y = 2 in                              let y = 2 in
        (x + 2) * (y - 2) in                      0 in
    b                                         b
```

It is clear that the expression `(x + 2) * (y - 2)` has been optimized to zero, as `y = 2` and `(y-2) = 0`. Going further with `-p`, we get:

```
fun int main() =
    let a = read(int) in
    let b =                               fun int main() =
        let x = a in             →            let a = read(int) in
        let y = 2 in                          let b = 0 in
        (x + 2) * (y - 2) in                  b
    b
```

Here the dead variables, `x` and `y`, have also been removed.
This refers to **??**

A Interpreter AND and OR

```
| evalExp ( And(e1, e2, pos), vtab, ftab ) =
      let val r1 = evalExp(e1, vtab, ftab)
      in case r1 of
         BoolVal b1 => if b1 then
                        let val r2 = evalExp(e2, vtab, ftab)
                        in case r2 of
                           BoolVal b2 => BoolVal b2
                         | otherwise  => raise Error ("And expect boolval", pos)
                        end
                       else BoolVal b1
       | otherwise  => raise Error ("And expect boolval", pos)
      end

| evalExp ( Or(e1, e2, pos), vtab, ftab ) =
      let val r1 = evalExp(e1, vtab, ftab)
      in case r1 of
         BoolVal b1 => if not b1 then
                        let val r2 = evalExp(e2, vtab, ftab)
                        in case r2 of
                           BoolVal b2 => BoolVal b2
                         | otherwise  => raise Error ("Or expect boolval", pos)
                        end
                       else BoolVal b1
       | otherwise  => raise Error ("Or expect boolval", pos)
      end
```

B Code Generator AND and OR

```
| And (e1, e2, pos) =>
    let val falseLabel = newName "falseLabel"
        val endLabel   = newName "endLabel"
        val t1         = newName "and_L"
        val t2         = newName "and_R"
        val code1      = compileExp e1 vtable t1
        val code2      = compileExp e2 vtable t2
    in code1                               @
       [Mips.BEQ (t1, "0", falseLabel)] @
       code2                               @
       [Mips.BEQ (t2, "0", falseLabel)] @
       [Mips.LI (place, "1")]              @
       [Mips.J endLabel]                   @
       [Mips.LABEL falseLabel]             @
       [Mips.LI (place, "0")]              @
       [Mips.LABEL endLabel]
     end
| Or (e1, e2, pos) =>
    let val trueLabel = newName "trueLabel"
        val endLabel  = newName "endLabel"
        val t1        = newName "or_L"
        val t2        = newName "or_R"
        val code1     = compileExp e1 vtable t1
        val code2     = compileExp e2 vtable t2
    in code1                              @
       [Mips.BNE (t1, "0", trueLabel)] @
       code2                              @
       [Mips.BNE (t2, "0", trueLabel)] @
       [Mips.LI (place, "0")]             @
       [Mips.J endLabel]                  @
       [Mips.LABEL trueLabel]             @
       [Mips.LI (place, "1")]             @
       [Mips.LABEL endLabel]
    end
```

C Interpreter NOT

```
| evalExp ( Not(e1, pos), vtab, ftab ) =
      let val r1 = evalExp(e1, vtab, ftab)
      in case r1 of
          BoolVal true  => BoolVal false
        | BoolVal false => BoolVal true
        | other         => raise Error("Not expects a boolean value", pos)
      end
```

D Code Generation NOT

```
| Not (e1, pos) =>
    let val zeroLabel = newName "zeroLabel"
        val endLabel  = newName "endLabel"
        val t1        = newName "not_R"
        val code      = compileExp e1 vtable t1
    in code                              @
       [Mips.BEQ (t1, "0", zeroLabel)] @
       [Mips.XOR (place, t1, t1)]        @
       [Mips.J endLabel]                 @
       [Mips.LABEL zeroLabel]            @
       [Mips.LI (place, "1")]            @
       [Mips.LABEL endLabel]
    end
```

E Boolean Implementation

E.1 Lexer

```
| "true"          => Parser.BOOLLIT (true, pos)
| "false"         => Parser.BOOLLIT (false , pos)
```

E.2 Parser

```
(...)
%token <bool*(int*int)> BOOLLIT
(...)
| BOOLLIT                 { Constant (BoolVal (#1 $1), #2 $1) }
```

E.3 Interpreter

```
fun evalExp ( Constant (v,_), vtab, ftab ) = v
```

E.4 typechecker

```
In.Constant  (v, pos)      => (valueType v, Out.Constant (v, pos))
```

E.5 Code Generation

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