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 10, 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
```

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 two new functions `evalAnd` and `evalOr` which evaluated and and or operations and returned an error if the expression to the left or right of the operators didn't return a boolean value.

```
fun evalAnd (BoolVal e1, BoolVal e2, pos) = BoolVal (e1 andalso e2)
  | evalAnd (e1, e2, pos) = raise Error("&& expects bool operands", pos)

fun evalOr (BoolVal e1, BoolVal e2, pos) = BoolVal (e1 orelse e2)
  | evalOr (e1, e2, pos) = raise Error("|| expects bool operands", pos)
```

We evaluated the expressions to the left and right of the operator and then called the helper functions afterwards.

In the code generator it was 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,

```
;; Code computing the left hand side of an and, and saving result to $t1

beq $t1, $zero, False

;; Code computing the right hand side of an and, and saving the result to $t2

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.

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`: $(\alpha \to \text{bool}) * [\alpha] \to [\alpha]$, 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: $t_{in} \to t_{out}$ for some types t_{in} and t_{out}.
- `IF` $t_{in} = t_e$ AND $t_{out} =$ `bool THEN filter(f, x) ELSE error()`.

`scan`: $(\alpha * \alpha \to \alpha) * \alpha * [\alpha] \to [\alpha]$, 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 λ-expressions in SOAC's

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`. 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.

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

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
```