Oversættere - Gruppeprojekt

Nicolai Nebel Jørgensen
Sofie Aleksandra Borup Harning
Rasmus Friis

December 21, 2014

1 Task 1

First of all, we extended Lexer.lex with the required symbols. Every token carries the position along with it. Boolean values get their value along as well.

```
fun keyword s =
   ...
 | "true"         => Parser.BOOLLIT (true, pos)
 | "false"        => Parser.BOOLLIT (false, pos)
 | "not"          => Parser.NOT pos

rule token = parse
   ...
 | `~`                { Parser.NEGATE (getPos lexbuf) }

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

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

We added two things to the mosmlyac parser specification in Parser.grm. First, we defined associativity and precedence for the new operators. Logical operators have lower precedence than comparison and integer operators.

```
\%nonassoc ifprec letprec
\%left OR
\%left AND
\%right NOT
\%left DEQ LTH
\%left PLUS MINUS
\%left TIMES DIVIDE
\%right NEGATE
```

We then added rules for parsing the different operators. This was basically copying the grammar from the assignment. Ambiguities were already solved through the associativity and precedence declarations.

Boolean values were parsed from the BOOLLIT token.

```
 | BOOLLIT             { Constant (BoolVal (#1 $1), #2 $1)  }
 | Exp TIMES Exp       { Times($1, $3, $2)  }
 | Exp DIVIDE Exp      { Divide($1, $3, $2) }
 | Exp AND Exp         { And($1, $3, $2)    }
```

```
| Exp OR Exp          { Or($1, $3, $2)     }
| Exp DEQ   Exp       { Equal($1, $3, $2)  }
| Exp LTH   Exp       { Less ($1, $3, $2)  }
| NOT Exp             { Not($2, $1)        }
| NEGATE Exp          { Negate($2, $1)     }
```

Next came the interpreter changes. Times and Divide were easily implemented using almost the same code as Plus, simply changing "op+" to "op *" and "op/" in the call to evalBinOpNum. Here is the code for Times, Divide is similar.

```
    | 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
```

And and Or is done short-circuiting by evaluating the first expression and then checking if we are ready to shirt-circuit. If we aren't, the evaluation continues with the second expression.

This implementation doesn't use the build-in short-circuiting andalso and orelse in SML. It seems like this would be a more elegant solution, but we couldn't figure out a good way to do it.

```
    | evalExp ( And(e1, e2, pos), vtab, ftab) =
          let val cond = evalExp(e1, vtab, ftab)
          in case cond of
                  BoolVal false => BoolVal false
                | BoolVal true  => evalExp(e2, vtab, ftab)
                | _             => raise Error("Argument to &&
                                    was not a boolean", pos)
          end
```

Not and negate are simply implemented using the SML not and " ".

```
    | evalExp ( Not(e1, pos), vtab, ftab) =
          (case evalExp(e1, vtab, ftab) of
              BoolVal b => BoolVal (not b)
            | _         => raise Error("Tried to not a
                                                                  non-boolean", pos)
           )
```

For the typechecker, the binary operators were again much the same as the implementation for 'Plus'. In this case, it included the logical operators as well. Times and And look very much like Plus:

```
| 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.And (e1, e2, pos)
     => let val (_, e1_dec, e2_dec) =
                    checkBinOp ftab vtab (pos, Bool, e1, e2)
        in (Bool,
            Out.And (e1_dec, e2_dec, pos))
        end
```

The implementation of 'times' is barely different, while And has the Int types replaced with Bool.

Not and Negate only needs to check one parameter, so the implementation just calls CheckExp. Negate needs an Int and Not needs a Bool.

```
| In.Not (e1, pos)
   => let val (t, e_dec) = checkExp ftab vtab e1
      in
        if t = Bool
        then
          (t,
           Out.Not (e_dec, pos))
        else raise Error ("Not: Argument is not a bool", pos)
      end
```

The next step is code generation. The first thing to do was to add support for boolean values. A reasonable representation seemed to be 0 for false and 1 for true. The following 'Constant' patters were added in compileExp:

```
    | Constant (BoolVal false, pos) => [ Mips.LI (place, makeConst 0) ]
    | Constant (BoolVal true, pos) => [Mips.LI (place, makeConst 1) ]
```

The code for the different is pretty similar. Compile the individual expressions and add a few lines of code that operate on the results. Times, for example just uses the Mips instruction mul to compute the result:

```
    | 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 looks the same, but uses div instead of mul to compute. Negate only has one expression to compile beforehand, but otherwise is the same. It is implemented by subtracting the result from 0.

Not is a bit more interesting. The result is found by xor-ing the result by 1. If the expression compiled to true, it would be $1 xor 1 = 0$ and if it was false it would get $0 xor 1 = 1$.

```
      | Not (e1, pos) =>
          let
            val t1 = newName "not"
            val code = compileExp e1 vtable t1
          in
            code @ [ Mips.XORI (place, t1, "1")
                   ]
          end
```

The logical operators And and Or needed to be short-circuiting. For And, the code looks like this:

```
      | And (e1, e2, pos) =>
          let
            (...)
          in
            code1
            @ [Mips.BEQ(place, "0", end_loop)]
            @ code2
            @ [Mips.LABEL(end_loop)]
          end
```

First, code1 is evaluated. If it is false, we already have the answer and branch out. Note that the result is already in place, so we don't need to move it. code2 is then evaluated, and it's result is put in place. Whatever it evaluates to will be the result. Or is implemented the same way, but it branches if code1 returns true.

2 Task 2

```
      | Scan (binop, acc_exp, arr_exp, tp, pos) =>
          let
```

```
            (* Name definitions *)
            (...)

        (* Compile array code *)
        val arr_code = compileExp arr_exp vtable arr_reg
        val header1 = [ Mips.LW (size_reg, arr_reg, "0")
                      , Mips.ADDI (size_reg, size_reg, "1") ]

        (* Compile initial value of acc, place into acc_reg *)
        val acc_code = compileExp acc_exp vtable acc_reg

        (* Make regs point to first element in arrays, i_reg = 0 *)
        val init_regs =
          [ Mips.ADDI (arr_reg, arr_reg, "4")
          , Mips.MOVE (i_reg, "0")
          , Mips.ADDI (addr_reg, place, "4")
          ]


        (* loop code *)
        val loop_header =
          [ Mips.LABEL(loop_beg)
          , Mips.SUB(tmp_reg, i_reg, size_reg)
          , Mips.BGEZ(tmp_reg, loop_end)
          ]

        (* Load arr[i] into tmp_reg *)
        val load_code =
              case getElemSize tp of
                  One =>  [ Mips.LB   (tmp_reg, arr_reg, "0")
                          , Mips.ADDI (arr_reg, arr_reg, "1") ]
                | Four => [ Mips.LW   (tmp_reg, arr_reg, "0")
                          , Mips.ADDI (arr_reg, arr_reg, "4") ]

        (* place := binop(tmp_reg, place) *)
        val apply_code =
              applyFunArg(binop, [acc_reg, tmp_reg], vtable, acc_reg, pos)

        (* Load arr[i] into tmp_reg *)
        val store_code =
              case getElemSize tp of
                  One =>  [ Mips.SB   (acc_reg, addr_reg, "0")
                          , Mips.ADDI (addr_reg, addr_reg, "1") ]
                | Four => [ Mips.SW   (acc_reg, addr_reg, "0")
                          , Mips.ADDI (addr_reg, addr_reg, "4") ]
        val loop_footer =
```

```
            [ Mips.ADDI (i_reg, i_reg, "1")
            , Mips.J (loop_beg)
            , Mips.LABEL (loop_end)
            ]
        in
          arr_code
          @ header1
          @ acc_code
          @ dynalloc(size_reg, place, tp)
          @ init_regs
          @ store_code
          @ loop_header
          @ load_code
          @ apply_code
          @ store_code
          @ loop_footer
        end
```

```
```

3 Task 3

4 Task 4

```
```