Aflevering 3

Thomas Vinther, 201303874 Jens Kristian Nielsen, 201303862

18. februar 2019

Opgave 36

```
def unparse(n: AstNode): String = n match {
  /** simple expression cases */
  case IntLit(c) => c.toString
  case BoolLit(c) => c.toString
  case VarExp(x) => s"$x"
  case StringLit(c) => c.toString
  case FloatLit(c) => c.toString

  /** combined expression cases*/
  case BinOpExp(leftexp, op, rightexp) =>
    val left = unparse(leftexp)
    val right = unparse(rightexp)
    val op1 = unparse(op)
    "("+left + op1 + right +")"
  case UnOpExp(op, exp) =>
    val op1 = unparse(op)
    val exp1 = unparse(exp)
    op1+"("+exp1+")"
  case BlockExp(vals,exp) =>
    var valString = ""
    var endTuborg = ""
    for(d <- vals){
      valString += " { " + unparse(d) + ";"
      endTuborg = endTuborg+" } "
    }
    valString+unparse(exp)+endTuborg
  case IfThenElseExp(conditionexp,thenexp,elseexp) =>
    val condi = unparse(conditionexp)
    val thene = unparse(thenexp)
    val elsee = unparse(elseexp)
    "if( "+condi+" ) "+thene+" else "+elsee
  case MatchExp(expr,caseList) =>
    unparse(expr)+"match"+caseList.map(unparse).mkString("{",";","}")

  case TupleExp(exps) =>
    exps.map(unparse).mkString("(",",",")")

  /** operator cases */
  case PlusBinOp()=> "+"
  case MinusBinOp() => "-"
  case DivBinOp() => "/"
  case MultBinOp() => "*"
  case ModuloBinOp() => "%"
  case MaxBinOp() => "max"
  case AndBinOp() => "&"
  case OrBinOp() => "|"
  case EqualBinOp() => "=="
  case LessThanBinOp() => "<"
  case LessThanOrEqualBinOp() => "<="
```

```
50      case NegUnOp() => "-"
51      case NotUnOp() => "!"
52
53      /** declarations */
54      case ValDecl(x,opttype,expr) =>
55      if(opttype.isDefined){
56        s"val $x ; "+unparse(opttype.get)+" = "+unparse(expr)}
57      else s"val $x = "+unparse(expr)
58
59      /** types */
60      case IntType() => "Int"
61      case BoolType() => "Boolean"
62      case FloatType() => "Float"
63      case StringType() => "String"
64      case TupleType(Nil) => "Unit"
65      case TupleType(list) =>
66        list.map(unparse).mkString("(",",",")")
67      case MatchCase(vars,e) =>
68        vars.mkString("(",",",") => ")+unparse(e)
69    }
```

Opgave 37

```
1  def eval(e: Exp, env: VarEnv): Val = e match {
2    case IntLit(c) =>
3      trace("Integer "+c+ " found")
4      IntVal(c)
5    case BoolLit(c) =>
6      trace("Boolean "+c+ "found")
7      BoolVal(c)
8    case FloatLit(c) =>
9      trace("Float"+c+ "found")
10     FloatVal(c)
11   case StringLit(c) =>
12     trace("String \""+c+ "\" found")
13     StringVal(c)
14   case VarExp(x) =>
15     trace(s"Variable $x found, lookup of variable value in environment gave "+venv(
         x))
16     env.getOrElse(x, throw new InterpreterError(s"Unknown identifier '$x'", e))
17   case BinOpExp(leftexp, op, rightexp) =>
18     trace("BinOpExp found, evaluating left and right expressions")
19     val leftval = eval(leftexp, env)
20     val rightval = eval(rightexp, env)
21     op match {
22       case PlusBinOp() => trace("Adding expressions")
23         (leftval,rightval) match{
24           case (IntVal(a),IntVal(b)) => IntVal(a+b)
25           case (FloatVal(a),IntVal(b)) => FloatVal(a+b)
26           case (IntVal(a),FloatVal(b)) => FloatVal(a+b)
27           case (StringVal(a),StringVal(b)) => StringVal(a+b)
28           case (StringVal(a),IntVal(b)) => StringVal(a+b)
29           case (StringVal(a),FloatVal(b)) => StringVal(a+b)
30           case (IntVal(a),StringVal(b)) => StringVal(a+b)
31           case (FloatVal(a),StringVal(b)) => StringVal(a+b)
32           case _ => throw new InterpreterError("Illegal addition",e)
33         }
34       case MinusBinOp() =>
35         trace("Subtracting expressions")
36         (leftval,rightval) match{
37           case (IntVal(a),IntVal(b)) => IntVal(a-b)
```

```
38            case (FloatVal(a),IntVal(b)) => FloatVal(a-b)
39            case (IntVal(a),FloatVal(b)) => FloatVal(a-b)
40            case (FloatVal(a),FloatVal(b)) => FloatVal(a-b)
41            case _ => throw new InterpreterError("Illegal subtraction",e)
42        }
43        case MultBinOp() =>
44          trace("Multiplying expressions")
45          (leftval,rightval) match{
46            case (IntVal(a),IntVal(b)) => IntVal(a*b)
47            case (FloatVal(a),IntVal(b)) => FloatVal(a*b)
48            case (IntVal(a),FloatVal(b)) => FloatVal(a*b)
49            case (FloatVal(a),FloatVal(b)) => FloatVal(a*b)
50            case _ => throw new InterpreterError("Illegal multiplication",e)
51          }
52        case DivBinOp() =>
53          if (rightval == IntVal(0) || rightval == FloatVal(0.0f))
54            throw new InterpreterError(s"Division by zero", op)
55          trace("Dividing expressions")
56          (leftval,rightval) match{
57            case (IntVal(a),IntVal(b)) => IntVal(a/b)
58            case (FloatVal(a),IntVal(b)) => FloatVal(a/b)
59            case (IntVal(a),FloatVal(b)) => FloatVal(a/b)
60            case (FloatVal(a),FloatVal(b)) => FloatVal(a/b)
61            case _ => throw new InterpreterError("Illegal division",e)
62          }
63        case ModuloBinOp() =>
64          if(rightval == IntVal(0) || rightval == FloatVal(0.0f)){throw new
                InterpreterError("Modulo by zero",op)}
65          trace("Calculating modulo")
66          (leftval,rightval) match{
67            case (IntVal(a),IntVal(b)) => IntVal(a%b)
68            case (FloatVal(a),IntVal(b)) => FloatVal(a%b)
69            case (IntVal(a),FloatVal(b)) => FloatVal(a%b)
70            case (FloatVal(a),FloatVal(b)) => FloatVal(a%b)
71            case _ => throw new InterpreterError("Illegal modulation",e)
72          }
73        case MaxBinOp() =>
74          trace("Finding max of expressions")
75          (leftval,rightval) match{
76            case (IntVal(a),IntVal(b)) => if(a>b){IntVal(a)}else{IntVal(b)}
77            case (FloatVal(a),IntVal(b)) => if(a>b){FloatVal(a)}else{FloatVal(b)}
78            case (IntVal(a),FloatVal(b)) => if(a>b){FloatVal(a)}else{FloatVal(b)}
79            case (FloatVal(a),FloatVal(b)) => if(a>b){FloatVal(a)}else{FloatVal(b)}
80            case _ => throw new InterpreterError("Illegal maksium",e)
81          }
82        case EqualBinOp()=>
83          trace("Evaluating equal")
84          (leftval,rightval) match {
85            case(IntVal(a),IntVal(b)) => BoolVal(a==b)
86            case(FloatVal(a),IntVal(b))=> BoolVal(a==b)
87            case(IntVal(a),FloatVal(b))=> BoolVal(a==b)
88            case(FloatVal(a),FloatVal(b))=> BoolVal(a==b)
89            case(StringVal(a),StringVal(b))=> BoolVal(a==b)
90            case(BoolVal(a),BoolVal(b)) => BoolVal(a==b)
91            case(TupleVal(a),TupleVal(b)) => BoolVal(a==b)
92            case _=> BoolVal(false)
93          }
94        case LessThanBinOp()=>
95          trace("Evaluating less than")
96          (leftval,rightval) match {
97            case (IntVal(a),IntVal(b)) => BoolVal(a<b)
98            case(FloatVal(a),IntVal(b))=> BoolVal(a<b)
99            case(IntVal(a),FloatVal(b))=> BoolVal(a<b)
```

```
100           case (FloatVal(a),FloatVal(b))=> BoolVal(a<b)
101           case _=> throw new InterpreterError("Illegal less than operation",op)
102         }
103       case LessThanOrEqualBinOp()=>
104         trace("Evaluating less than or equal")
105         (leftval,rightval) match {
106           case (IntVal(a),IntVal(b)) => BoolVal(a<=b)
107           case(FloatVal(a),IntVal(b))=> BoolVal(a<=b)
108           case(IntVal(a),FloatVal(b))=> BoolVal(a<=b)
109           case (FloatVal(a),FloatVal(b))=> BoolVal(a<=b)
110           case _=> throw new InterpreterError("Illegal 'less than or equal'
                operation",op)
111         }
112       case AndBinOp()=>
113         trace("Evaluating less than or equal")
114           (leftval,rightval) match {
115           case (BoolVal(a),BoolVal(b)) => BoolVal(a&b)
116           case _=> throw new InterpreterError("Illegal 'and' operation",op)
117         }
118       case OrBinOp()=>
119         trace("Evaluating less than or equal")
120           (leftval,rightval) match {
121           case (BoolVal(a),BoolVal(b)) => BoolVal(a|b)
122           case _=> throw new InterpreterError("Illegal 'and' operation",op)
123           }
124     }
125     case UnOpExp(op, exp) =>
126       trace("Unary expression found")
127       val expval = eval(exp, env)
128       op match {
129         case NegUnOp() =>
130           trace("Negation of number")
131           expval match{
132             case IntVal(a) => IntVal(-a)
133             case FloatVal(a) => FloatVal(-a)
134             case _ => throw new InterpreterError("Not a number",e)
135           }
136         case NotUnOp() =>
137           trace("Negation of Boolean")
138           expval match{
139             case BoolVal(a) => BoolVal(!a)
140             case _ => throw new InterpreterError("Not a Boolean",e)
141           }
142       }
143     case IfThenElseExp(condexp, thenexp, elseexp) =>
144       eval(condexp,env) match {
145         case BoolVal(a) =>
146           trace("If statement found, evaluating condition")
147           if (a) {
148             trace("evaluating then clause")
149             return eval(thenexp, env)
150           } else trace("evaluationg else clause")
151           eval(elseexp, env)
152         case _ => throw new InterpreterError("Condition clause not a boolean",
              IfThenElseExp(condexp, thenexp, elseexp))
153       }
154     case BlockExp(vals, exp) =>
155       var env1 = env
156       trace("Calculating variable values and adding to variable environment")
157       for (d <- vals) {
158         val dexp = eval(d.exp,env1)
159         checkValueType(dexp, d.opttype, d)
160         env1 += (d.x -> dexp)
```

```
      }
      eval(exp, env1)
    case TupleExp(exps) =>
      trace("Evaluation tuple of expressions")
      var vals = List[Val]()
      for (ex <- exps)
        vals = eval(ex, env) :: vals
        TupleVal(vals.reverse)
    case MatchExp(exp, cases) =>
      trace("Updating ")
      val expval = eval(exp, env)
      expval match {
        case TupleVal(vs) =>
          for (c <- cases) {
          if (vs.length == c.pattern.length) {
            val venv_update = c.pattern.zip(vs)
            return eval(c.exp,env++venv_update)
          }
      }
    throw new InterpreterError(s"No case matches value ${valueToString(expval)}", e
        )
    case _ => throw new InterpreterError(s"Tuple expected at match, found ${
        valueToString(expval)}", e)
    }
}
```

Opgave 38

```
def typeCheck(e: Exp, vtenv: VarTypeEnv): Type = e match {
  case IntLit(_) => IntType()
  case BoolLit(_) => BoolType()
  case FloatLit(_) => FloatType()
  case StringLit(_) => StringType()
  case VarExp(x) => vtenv.getOrElse(x, throw new TypeError(s"Unknown identifier '$x
      '", e))
  case BinOpExp(leftexp, op, rightexp) =>
    val lefttype = typeCheck(leftexp, vtenv)
    val righttype = typeCheck(rightexp, vtenv)
    op match {
      case PlusBinOp() =>
        (lefttype, righttype) match {
          case (IntType(), IntType()) => IntType()
          case (FloatType(), FloatType()) => FloatType()
          case (IntType(), FloatType()) => FloatType()
          case (FloatType(), IntType()) => FloatType()
          case (StringType(), StringType()) => StringType()
          case (StringType(), IntType()) => StringType()
          case (StringType(), FloatType()) => StringType()
          case (IntType(), StringType()) => StringType()
          case (FloatType(), StringType()) => StringType()
          case _ => throw new TypeError(s"Type mismatch at '+', unexpected types ${
              unparse(lefttype)} and ${unparse(righttype)}", op)
        }
      case MinusBinOp() | MultBinOp() | DivBinOp() | ModuloBinOp() | MaxBinOp() =>
      (lefttype, righttype) match {
        case (IntType(), IntType()) => IntType()
        case (FloatType(), FloatType()) => FloatType()
        case (IntType(), FloatType()) => FloatType()
        case (FloatType(), IntType()) => FloatType()
        case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
            unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op)
```

```
31        }
32        case EqualBinOp() => BoolType()
33        case LessThanBinOp() | LessThanOrEqualBinOp() =>
34          (lefttype, righttype) match{
35            case (IntType(),IntType()) => BoolType()
36            case (FloatType(), FloatType()) => BoolType()
37            case (IntType(), FloatType()) => BoolType()
38            case (FloatType(), IntType()) => BoolType()
39            case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                  unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op)
40          }
41        case AndBinOp() | OrBinOp() =>
42        (lefttype, righttype) match{
43          case (BoolType(),BoolType()) => BoolType()
44          case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op)
45        }
46      }
47    case UnOpExp(op, exp) => op match{
48      case NegUnOp() =>
49        typeCheck(exp,vtenv) match{
50        case IntType() => IntType()
51          case FloatType() => FloatType()
52          case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                unexpected type ${unparse(typeCheck(exp,vtenv))}}", op)
53        }
54      case NotUnOp() =>
55        typeCheck(exp,vtenv) match{
56          case BoolType() => BoolType()
57        case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}', unexpected
               type ${unparse(typeCheck(exp,vtenv))}}", op)
58        }
59      }
60    case IfThenElseExp(condexp, thenexp, elseexp) =>
61      val ce = typeCheck(condexp,vtenv)
62      val te = typeCheck(thenexp,vtenv)
63      val ee = typeCheck(elseexp,vtenv)
64      (ce,te,ee) match{
65        case (BoolType(),IntType(),IntType()) => IntType()
66        case (BoolType(),FloatType(),FloatType()) => FloatType()
67        case (BoolType(),StringType(),StringType()) => StringType()
68        case (BoolType(),BoolType(),BoolType()) => BoolType()
69        case _ => throw new TypeError(s"Type mismatch at If statement, unexpected
             type either in the condition ${unparse(ce)} or in the inner expressions
             that must be of the same type ${unparse(te)} = ${unparse(ee)}",
             IfThenElseExp(condexp, thenexp, elseexp))
70    }
71    case BlockExp(vals, exp) =>
72      var vtenv1 = vtenv
73    for (d <- vals) {
74        val t = typeCheck(d.exp, vtenv1)
75        checkTypesEqual(t, d.opttype, d)
76        vtenv1 = vtenv1 + (d.x -> d.opttype.getOrElse(t))
77      }
78      typeCheck(exp,vtenv1)
79    case TupleExp(exps) => TupleType(exps.map(x => typeCheck(x,vtenv)))
80    case MatchExp(exp, cases) =>
81      val exptype = typeCheck(exp, vtenv)
82      exptype match {
83        case TupleType(ts) =>
84        for (c <- cases) {
85            if (ts.length == c.pattern.length) {
86              val venv_update = c.pattern.zip(ts)
```

```
87              return typeCheck(c.exp,vtenv++venv_update)
88            }
89      }
90    throw new TypeError(s"No case matches type ${unparse(exptype)}", e)
91    case _ => throw new TypeError(s"Tuple expected at match, found ${unparse(exptype)
          }", e)
92    }
93  }
```