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 _ => throw new InterpreterError("Illegal subtraction",e)
41          }
42          case MultBinOp() =>
43            trace("Multiplying expressions")
44            (leftval,rightval) match{
45              case (IntVal(a),IntVal(b)) => IntVal(a*b)
46              case (FloatVal(a),IntVal(b)) => FloatVal(a*b)
47              case (IntVal(a),FloatVal(b)) => FloatVal(a*b)
48              case _ => throw new InterpreterError("Illegal multiplication",e)
49            }
50          case DivBinOp() =>
51            if (rightval == IntVal(0) || rightval == FloatVal(0.0f))
52              throw new InterpreterError(s"Division by zero", op)
53            trace("Dividing expressions")
54            (leftval,rightval) match{
55              case (IntVal(a),IntVal(b)) => IntVal(a/b)
56              case (FloatVal(a),IntVal(b)) => FloatVal(a/b)
57              case (IntVal(a),FloatVal(b)) => FloatVal(a/b)
58              case _ => throw new InterpreterError("Illegal division",e)
59            }
60          case ModuloBinOp() =>
61            if(rightval == IntVal(0) || rightval == FloatVal(0.0f)){throw new
                 InterpreterError("Modulo by zero",op)}
62            trace("Calculating modulo")
63            (leftval,rightval) match{
64              case (IntVal(a),IntVal(b)) => IntVal(a%b)
65              case (FloatVal(a),IntVal(b)) => FloatVal(a%b)
66              case (IntVal(a),FloatVal(b)) => FloatVal(a%b)
67              case _ => throw new InterpreterError("Illegal modulation",e)
68            }
69          case MaxBinOp() =>
70          trace("Finding max of expressions")
71          (leftval,rightval) match{
72            case (IntVal(a),IntVal(b)) => if(a>b){IntVal(a)}else{IntVal(b)}
73            case (FloatVal(a),IntVal(b)) => if(a>b){FloatVal(a)}else{IntVal(b)}
74            case (IntVal(a),FloatVal(b)) => if(a>b){IntVal(a)}else{FloatVal(b)}
75            case _ => throw new InterpreterError("Illegal maksium",e)
76          }
77        }
78        case UnOpExp(op, exp) =>
79          trace("Unary expression found")
80          val expval = eval(exp, env)
81          op match {
82            case NegUnOp() =>
83              trace("Negation of number")
84              expval match{
85                case IntVal(a) => IntVal(-a)
86                case FloatVal(a) => FloatVal(-a)
87                case _ => throw new InterpreterError("Not a number",e)
88              }
89            case NotUnOp() =>
90              trace("Negation of Boolean")
91              expval match{
92                case BoolVal(a) => BoolVal(!a)
93                case _ => throw new InterpreterError("Not a Boolean",e)
94              }
95          }
96        case IfThenElseExp(condexp, thenexp, elseexp) =>
97          eval(condexp,env) match {
98            case BoolVal(a) =>
99              trace("If statement found, evaluating condition")
```

```
100           if (a) {
101             trace("evaluating then clause")
102             eval(thenexp, env)
103           } else trace("evaluationg else clause")
104           eval(elseexp, env)
105         case _ => throw new InterpreterError("Condition clause not a boolean",
                IfThenElseExp(condexp, thenexp, elseexp))
106       }
107     case BlockExp(vals, exp) =>
108       var env1 = env
109       trace("Calculating variable values and adding to variable environment")
110       for (d <- vals) {
111         val dexp = eval(d.exp,env1)
112         checkValueType(dexp, d.opttype, d)
113         env1 += (d.x -> dexp)
114       }
115       eval(exp, env1)
116     case TupleExp(exps) =>
117       trace("Evaluation tuple of expressions")
118       var vals = List[Val]()
119       for (ex <- exps)
120         vals = eval(ex, env) :: vals
121         TupleVal(vals.reverse)
122     case MatchExp(exp, cases) =>
123       trace("Updating ")
124       val expval = eval(exp, env)
125       expval match {
126         case TupleVal(vs) =>
127           for (c <- cases) {
128           if (vs.length == c.pattern.length) {
129             val venv_update = c.pattern.zip(vs)
130             return eval(c.exp,env++venv_update)
131           }
132       }
133     throw new InterpreterError(s"No case matches value ${valueToString(expval)}", e
          )
134     case _ => throw new InterpreterError(s"Tuple expected at match, found ${
          valueToString(expval)}", e)
135     }
136 }
```

Opgave 38

```
1  def typeCheck(e: Exp, vtenv: VarTypeEnv): Type = e match {
2    case IntLit(_) => IntType()
3    case BoolLit(_) => BoolType()
4    case FloatLit(_) => FloatType()
5    case StringLit(_) => StringType()
6    case VarExp(x) => vtenv.getOrElse(x, throw new TypeError(s"Unknown identifier '$x
         '", e))
7    case BinOpExp(leftexp, op, rightexp) =>
8      val lefttype = typeCheck(leftexp, vtenv)
9      val righttype = typeCheck(rightexp, vtenv)
10     op match {
11       case PlusBinOp() =>
12         (lefttype, righttype) match {
13           case (IntType(), IntType()) => IntType()
14           case (FloatType(), FloatType()) => FloatType()
15           case (IntType(), FloatType()) => FloatType()
16           case (FloatType(), IntType()) => FloatType()
17           case (StringType(), StringType()) => StringType()
```

```
18            case (StringType(), IntType()) => StringType()
19            case (StringType(), FloatType()) => StringType()
20            case (IntType(), StringType()) => StringType()
21            case (FloatType(), StringType()) => StringType()
22            case _ => throw new TypeError(s"Type mismatch at '+', unexpected types ${
                  unparse(lefttype)} and ${unparse(righttype)}", op)
23          }
24        case MinusBinOp() | MultBinOp() | DivBinOp() | ModuloBinOp() | MaxBinOp() =>
25        (lefttype, righttype) match {
26          case (IntType(), IntType()) => IntType()
27          case (FloatType(), FloatType()) => FloatType()
28          case (IntType(), FloatType()) => FloatType()
29          case (FloatType(), IntType()) => FloatType()
30          case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op)
31        }
32        case EqualBinOp() => BoolType()
33        case LessThanBinOp() | LessThanOrEqualBinOp() =>
34          (lefttype, righttype) match{
35            case (IntType(),IntType()) => BoolType()
36            case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                  unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op)
37          }
38        case AndBinOp() | OrBinOp() =>
39        (lefttype, righttype) match{
40          case (BoolType(),BoolType()) => BoolType()
41          case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                unexpected types ${unparse(lefttype)} and ${unparse(righttype)}", op)
42        }
43      }
44    case UnOpExp(op, exp) => op match{
45      case NegUnOp() =>
46        typeCheck(exp,vtenv) match{
47        case IntType() => IntType()
48          case FloatType() => FloatType()
49          case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}',
                unexpected type ${unparse(typeCheck(exp,vtenv))}}", op)
50        }
51      case NotUnOp() =>
52        typeCheck(exp,vtenv) match{
53          case BoolType() => BoolType()
54        case _ => throw new TypeError(s"Type mismatch at '${unparse(op)}', unexpected
              type ${unparse(typeCheck(exp,vtenv))}}", op)
55        }
56      }
57    case IfThenElseExp(condexp, thenexp, elseexp) =>
58      val ce = typeCheck(condexp,vtenv)
59      val te = typeCheck(thenexp,vtenv)
60      val ee = typeCheck(elseexp,vtenv)
61      (ce,te,ee) match{
62        case (BoolType(),IntType(),IntType()) => IntType()
63        case (BoolType(),FloatType(),FloatType()) => FloatType()
64        case (BoolType(),StringType(),StringType()) => StringType()
65        case (BoolType(),BoolType(),BoolType()) => BoolType()
66        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())
67    }
68    case BlockExp(vals, exp) =>
69      var vtenv1 = vtenv
70    for (d <- vals) {
71        val t = typeCheck(d.exp, vtenv1)
```

```
72        checkTypesEqual(t, d.opttype, d)
73        vtenv1 = vtenv1 + (d.x -> d.opttype.getOrElse(t))
74      }
75      typeCheck(exp,vtenv1)
76    case TupleExp(exps) => TupleType(exps.map(x => typeCheck(x,vtenv)))
77    case MatchExp(exp, cases) =>
78      val exptype = typeCheck(exp, vtenv)
79      exptype match {
80        case TupleType(ts) =>
81        for (c <- cases) {
82            if (ts.length == c.pattern.length) {
83              val venv_update = c.pattern.zip(ts)
84              return typeCheck(c.exp,vtenv++venv_update)
85            }
86      }
87    throw new TypeError(s"No case matches type ${unparse(exptype)}", e)
88    case _ => throw new TypeError(s"Tuple expected at match, found ${unparse(exptype)
        }", e)
89    }
90  }
```